ISF

Masterarbeit

Integration von Evolution in die Modellierung und Analyse von Softwareproduktlinien

Sophia Nahrendorf

15.10.2017

-
Institut für Softwaretechnik und Fahrzeuginformatik
an der
Technischen Universität Carolo-Wilhelmina in Braunschweig

Prof. Dr. Ina Schaefer

Sascha Lity, M.Sc.

Zusammenfassung

Aufgrund ihrer Variabilität umfassen Softwareproduktlinien eine hohe Anzahl von vielfältigen, aber dennoch ähnlichen Produktvarianten. Diese Tatsache sowie die ausgeprägte Wiederverwendung von Softwarebausteinen, zwischen denen außerdem komplizierte Abhängigkeiten bestehen können, lassen die Evolution von Softwareproduktlinien äußerst komplex werden. Diese Komplexität kann zu Inkonsistenzen und Fehlern bei der Evolution führen. Zur Vermeidung dessen werden Modelle eingesetzt, damit bereits vor der eigentlichen Evolution die zukünftigen Veränderungen betrachtet und untersucht sowie eventuelle Probleme durch Auswirkungen der Evolution abgeschätzt werden können. Folglich werden Modellierungsmethoden benötigt, die nicht nur die Variabilität der Produktlinien übersichtlich und verständlich darstellen, sondern auch Evolution abbilden können und eine effiziente Analyse von Evolutionsauswirkungen erlauben.

Für die Darstellung von Variabilität existieren zahlreiche Ansätze aus den Modellierungskategorien der annotativen, kompositionalen und transformationalen Modelle. Die Anzahl der Techniken, die gleichzeitig mit Evolution umgehen können, sind jedoch weitaus geringer. Dies ist insofern problematisch, dass nicht für jede der genannten Kategorien Evolution auf eine einfache Weise modelliert werden kann. Die Kategorien bieten allerdings unterschiedliche Vor- und Nachteile bei der Darstellung und sind daher je nach Betrachtungsfokus ungleich nützlicher. Aus diesem Grund wird in dieser Arbeit mit der 175%-Modellierung eine Modellierungsmethode für Evolution in annotativen Konzepten entwickelt, da für diese Kategorie noch keine Methode existiert, die gleichermaßen normale Variabilität und Evolution handhaben kann. Damit eine flexible Anwendung von Ansätzen aus unterschiedlichen Modellierungskategorien, beispielsweise zum Vergleich, ermöglicht wird, wird außerdem ein Algorithmus zur Transformation von Higher-Order Delta-Modellen - einem transformationalen Ansatz - in 175%-Modelle entwickelt. So ist nur die Erstellung eines Modells nötig, um zwei Ansätze aus verschiedenen Kategorien verwenden zu können, deren Vor- und Nachteile sich gegenseitig ausgleichen können. Anhand von mathematischer Induktion wird außerdem bewiesen, dass der Algorithmus ein 175%-Modell erzeugt, das äquivalent zum eingegebenen Higher-Order Delta-Modell ist, d.h. dass sich aus beiden Modellen die gleichen Produktmodelle ableiten lassen. Da 175%-Modelle aufgrund der Kapselung sämtlicher Modellelemente in einem einzigen Modell sehr groß sind, wird darüber hinaus eine Möglichkeit zum Slicing von 175%-Modellen vorgestellt, um die Komplexität der Modelle reduzieren und diese auf Basis bestimmter Kriterien analysieren zu können. Sowohl die Technik zur 175%-Modellierung, der Transformationsalgorithmus und das 175%-Slicing werden als Eclipse-Plug-ins implementiert und die Implementierung in dieser Arbeit dokumentiert.

Stichwörter Softwareproduktlinien, Variabilitätsmodellierung, Evolution, 175%-Modellierung

Inhaltsverzeichnis

Inhaltsverzeichnis i

Abbildungsverzeichnis iii

Liste der Algorithmen v

1. Einleitung 1

2. Grundlagen 5
2.1. Softwareproduktlinien 5
2.2. Feature-annotierte State Machines 12
2.3. Slicing 15
2.4. Evolution von Softwareproduktlinien 22

3. Evolution von Feature-annotierten State Machines 35
3.1. Konzeption der Darstellung von 175%-Modellen 35
3.2. Formale Definition von 175%-Modellen 42
3.3. Slicing von 175%-Modellen 50

4. Transformation von Higher-Order Delta-Modellen in 175%-Modelle 55
4.1. Definition von Transformationsregeln 55
4.2. Algorithmus zur Transformation 71
4.3. Beweis des Transformationsalgorithmus 75
4.3.1. Äquivalenz Delta-Modell und 150%-Modell 76
4.3.2. Äquivalenz Higher-Order Delta-Modell und 175%-Modell 77
4.3.3. Induktionsbeweis 79

5. Implementierung 89
5.1. Anforderungen 89
5.2. Realisierung 91
5.2.1. Datenmodelle der bestehenden Implementierung 91
5.2.2. Datenmodelle der neuen Implementierung 99
5.2.3. Plug-in Architektur 102
5.3. 175%-Modelle und Versionsableitung 104
5.4. Modelltransformation 107
5.5. Erweiterung auf 175%-Slicing 113

6. Evaluation **119**
6.1. Fallstudien 121
6.2. Zeitverhalten der Funktionalitäten 123

7. Zusammenfassung und Fazit **133**

Literatur **141**

A. Verwendung der Eclipse Plug-ins **145**

B. Inhalt des Datenträgers **147**

Abbildungsverzeichnis

2.1. Framework der Softwareproduktlinienentwicklung (nach Pohl [33]) 6
2.2. Ein beispielhaftes Feature-Diagramm 8
2.3. Delta-Konzept am Beispiel einer State Machine [24] 11
2.4. State Machine eines Verkaufsautomaten 13
2.5. Feature-annotierte State Machine eines Verkaufsautomaten 14
2.6. State Machine eines Verkaufsautomaten vor und nach dem Slicing 18
2.7. Feature-annotierte State Machine (M, α) eines Verkaufsautomaten (angelehnt an [21]) 20
2.8. Slice von (M, α) für $\Gamma = \{cappuccino \mapsto false\}$ (angelehnt an [21]) 20
2.9. Feature-Modell eingeteilt in Fragmente 24
2.10. Evolutionsplan 25
2.11. Feature-Modelle eines Verkaufsautomaten 28
2.12. Temporales Feature-Modell eines Verkaufsautomaten 29
2.13. Hyper Feature-Modell des mobilen Roboters *TurtleBot* (nach Seidl et al. [36]) 31
2.14. Beispiel für Higher-Order Delta-Modellierung 33

3.1. Feature-Modell eines Verkaufsautomaten zu unterschiedlichen Zeitpunkten 36
3.2. Feature-annotierte State Machine eines Verkaufsautomaten zu verschiedenen Zeitpunkten 36
3.3. Versionierte Feature-annotierte State Machine des Verkaufsautomaten 37
3.4. Temporale Feature-annotierte State Machine 38
3.5. Konzept für 175%-Modelle am Beispiel eines Verkaufsautomaten 39
3.6. Konzept für Versionsfenster am Beispiel eines Verkaufsautomaten 40
3.7. Versionsfenster TSM_ϑ von TSM für $\vartheta = [\theta_3, \theta_6)$ 41
3.8. Temporales Feature-Modell vor und nach Erweiterung 46
3.9. 175%-Modell (M, α_{175}) für $\theta_n = \theta_1$ 46
3.10. 150%-Modelle für θ_1 und θ_2 47
3.11. 175%-Modell (M, α_{175}) 48
3.12. Temporale Feature-annotierte State Machine $M_{175} = (M, \alpha_{175})$ eines Verkaufsautomaten 51
3.13. Slice von M_{175} für $C_{175} = (\varepsilon, \{milk \mapsto false, dollar \mapsto false\}, \theta_7)$ 52

4.1. Beispiel-Kernmodell 56
4.2. Transformation eines Deltas mit hinzugefügtem Element 57
4.3. Transformation eines Deltas mit entferntem Element 58
4.4. Transformation eines Deltas mit modifiziertem Element 59
4.5. Transformation eines hinzugefügten Deltas mit hinzugefügtem Element 60
4.6. Transformation eines hinzugefügten Deltas mit entferntem Element 62

4.7. Transformation eines hinzugefügten Deltas mit modifiziertem Element 64
4.8. Transformation eines entfernten Deltas mit hinzugefügtem Element 65
4.9. Transformation eines entfernten Deltas mit entferntem Element 66
4.10. Transformation eines entfernten Deltas mit modifiziertem Element 67
4.11. Transformation eines modifizierten Deltas mit hinzugefügter Operation 68
4.12. Transformation eines modifizierten Deltas mit entfernter Operation 69
4.13. Transformation eines Deltas mit modifizierter Anwendungsbedingung 70
4.14. Produktableitung von Higher-Order Delta- und 175%-Modellen 78

5.1. Use-Case-Diagramm der Anforderungen 89
5.2. Arbeitsablauf und Zusammenspiel der verschiedenen Aufgaben 90
5.3. Meta-Modell Plug-in `de.imotep.core.datamodel` 92
5.4. Meta-Modell Plug-in `de.imotep.core.behavior` 93
5.5. Meta-Modell Plug-in `de.imotep.variability.annotatedBehavior` 94
5.6. Meta-Modell Plug-in `de.imotep.slicing` 95
5.7. Meta-Modell Plug-in `de.imotep.slicing.dependency` 96
5.8. Meta-Modell Plug-in `de.imotep.variability.deltaBehavior` 97
5.9. Meta-Modell Plug-in `de.imotep.dope` 98
5.10. Meta-Modell Plug-in `de.imotep.evolution.temporalAnnotation` 100
5.11. Meta-Modell Plug-in `de.imotep.evolution.transformation` 101
5.12. Ausschnitt des erweiterten Slicing-Meta-Modells 102
5.13. Architektur des Plug-ins 103
5.14. Wizard zur Erstellung eines 175%-Modells 104
5.15. Arbeitsablauf Versionsableitung 105
5.16. Aufrufen eines Wizard zur Versionsableitung 106
5.17. Wizard zur Ableitung einer Version aus einem 175%-Modell 107
5.18. Arbeitsablauf Transformation 108
5.19. 113

6.1. Vergleichsfaktoren für die 175%-Modelle der Fallstudien 122
6.2. Zeitverhalten Transformation 123
6.3. Zeitverhalten Versionsableitung 124
6.4. Zeitverhalten Versionsableitung für unterschiedliche Versionen 125
6.5. Zeitverhalten 150%-Slicing 126
6.6. Zeitverhalten 150%-Slicing ohne Body Comfort System 127
6.7. Laufzeit reine Slicing-Phase für verschiedene Versionen 127
6.8. Zeitverhalten 175%-Slicing 128
6.9. Zeitverhalten Vergleich 175%-Slicing und Kombination Versionsableitung - 150%-Slicing 129

7.1. Darstellung der Annotationen von 175%-Elementen im Editor 135

A.1. Ordnerstruktur Eclipse 145
A.2. Export eines Plug-ins 146

Liste der Algorithmen

2.1. Beispielprogramm P 17
2.2. Slice von P für $C = (10, sum)$ 17
2.3. Beispielprogramm Q 17
2.4. Bedingter Slice von Q für $C = (4, a, a > 0)$ 18
2.5. 150%-State Machine Slicing Algorithmus 21

3.1. 175%-State Machine Slicing Algorithmus 50

4.1. Algorithmus zur Transformation von HODs in 175%-Modelle 75

1 Einleitung

Softwareproduktlinien (SPL) [13, 33] sind aufgrund der Kombination von Massenproduktion mit der Erfüllung individueller Kundenwünsche in der Softwareentwicklung sehr verbreitet [13]. Sie bestehen aus einer Reihe von ähnlichen Produkten, die sich aus einem gemeinsamen Kern von Artefakten zusammensetzen, durch variable Artefakte aber dennoch unterschiedliche Ausprägungen zeigen. Die Realisierung von Softwareproduklinien kann angesichts von Abhängigkeiten zwischen den verschiedenen Artefakten äußerst komplex sein.

Damit bei der Entwicklung von SPLs und der Ableitung von Produkten Fehler und Inkonsistenzen frühzeitig entdeckt werden, werden Modelle verwendet [33]. Modelle helfen darüber hinaus dabei, die Übersicht über die SPL zu behalten, und können außerdem zu deren Analyse verwendet werden. Eine geeignete Modellierung von Softwareproduktlinien kann wegen deren Größe, Komplexität und Variabilität schwierig sein. Es gibt zudem unterschiedliche Möglichkeiten der Modellierung, um verschiedene Perspektiven der Produktlinie hervorzuheben.

Modellierung von Softwareproduktlinien

Feature-Modelle [22] werden beispielsweise dazu verwendet, Gemeinsamkeiten und Unterschiede von Softwareproduktlinien darzustellen. Hierbei werden wichtige Merkmale der SPL - sogenannte Features - und deren Abhängigkeiten untereinander in einer baumartigen Struktur festgehalten. Feature-Modelle definieren dabei klar, welche Features für alle Produkte gelten und auf welche Weise variable Features kombiniert werden dürfen. Eine gültige Kombination von Features wird Feature-Konfiguration genannt.

Des Weiteren existieren unterschiedliche Ansätze Variabilität in verschiedenen Arten von Produktmodellen (State Machines, Klassendiagramme, etc.) darzustellen [35]. Diese Ansätze lassen sich in die Kategorien kompositional, annotativ und transformational einteilen. Bei kompositionalen Ansätzen existieren kleinste Modellfragmente, denen bestimmte Features zugeordnet sind. Diese Fragmente werden dann je nach gewählter Feature-Konfiguration zu einem Produktmodell zusammengesetzt. Transformationale Methoden nutzen ein sogenanntes Kernmodell und wenden verschiedene Operationen darauf an. Die Operationen verändern das Kernmodell, sodass je nach Kombination unteschiedliche Produktmodelle daraus entstehen. Ein transformationaler Ansatz ist beispielsweise die Delta-Modellierung [10, 34]. Delta-Modelle setzen sich dabei aus einem Kernmodell und verschiedenen Deltas zusammen. Deltas bündeln Operationen, die häufig gemeinsam verwendet werden, und können bestimmten Features zugeordnet sein. Annotative Ansätze arbeiten mit Modellen, die sämtliche gemeinsamen und variablen Elemente aller möglichen Produktmodelle beinhalten. Die Elemente sind mit Informationen darüber annotiert, für welche Produktvarianten sie gelten. Solche Modelle werden unter anderem 150%-Modelle genannt und können beispielsweise mit Features annotiert sein [21]. Je nach Feature-Konfiguration werden nur bestimmte Modellinhalte benötigt und die entsprechenden Produktmodelle durch Entfernen der überflüssigen Elemente abgeleitet.

Mithilfe von Slicing [19, 39] können 150%-Modelle außerdem auf verschiedene Aspekte hin untersucht werden [19]. Slicing stammt ursprünglich aus der Programmanalyse und wird dazu genutzt, Programmteile zu entfernen, welche ein gewähltes Kriterium - beispielweise eine bestimmte Variable an einem bestimmten Programmpunkt - nicht beeinflussen oder nicht davon beeinflusst werden [39]. Auf Modelle angewendet, können somit jene Elemente eines Modells entfernt werden, die entsprechend des gewählten Kriteriums nicht relevant sind. Beim Conditioned Model Slicing [21] für Feature-annotierte Modelle können etwa bestimmte Elemente in Kombination mit teilweisen oder kompletten Feature-Konfigurationen als Slicing-Kriterium gewählt werden, um das Modell auf für die jeweilige Betrachtung relevante Modellinhalte zu reduzieren.

Insgesamt lassen sich Softwareproduktlinien somit durch Modelle und verschiedene Analysemethoden auf unterschiedliche Arten hinsichtlich ihrer Variabilität betrachten und analysieren. Zusätzlich zu der bereits beschriebenen Variabilität, können SPLs allerdings auch zeitlich variabel sein. Dies resultiert daraus, dass Softwareproduktlinien sich durch neue Anforderungen, die Notwendigkeit Fehler zu beheben oder auch Konkurrenz zwangsläufig weiterentwickeln [28].

Evolution von Softwareproduktlinien

Diese sogenannte Evolution [6, 27, 28] beschreibt die Veränderung von einer Version einer SPL in eine zeitlich darauffolgende Version. Zur Vermeidung von Fehlern und Inkonsistenzen während des Evolutionsprozesses ist eine geeignete Methodik zur Modellierung der Evolution notwendig. So können die Auswirkungen der Evolution bereits im Voraus abgeschätzt werden. Auch eine Betrachtung alter Versionen einer Produktlinie ist von Interesse, falls sich Produkte dieser alten Versionen noch im Umlauf befinden, welche gewartet oder repariert werden müssen. Daher ist es für eine effiziente Analyse während der Evolution förderlich, dass Modelle existieren, welche sowohl alte als auch neue Versionen einer SPL ineinander vereinen. Auf diese Weise entsteht eine Art Versionsüberblick beziehungsweise Evolutionshistorie in Form eines Modells. Für die zuvor beschriebene Delta-Modellierung existiert etwa Higher-Order Delta-Modellierung [25], um Evolution darzustellen. Dabei wird das komplette Delta-Modell verändert. Die Veränderungen werden durch Higher-Order Deltas beschrieben, welche komplette Deltas zum Delta-Modell hinzufügen, daraus entfernen oder sie modifizieren.

Allerdings reicht es nicht aus, nur eine Darstellungsvariante für Evolution in Softwareproduktlinien zu haben, da die verschiedenen Modellierungsansätze unterschiedliche Vor- und Nachteile bieten [35]. Für 150%-Modelle etwa existiert noch keine Möglichkeit Evolution darzustellen. Auch für diese Modellform ist es jedoch hilfreich, Evolutionshistorien festhalten zu können, um Analysen hinsichtlich der Evolution durchführen zu können. Daher muss die Methodik für 150%-Modelle erweitert werden, sodass die Modelle auch Evolution enthalten können. Mit den zusätzlichen Informationen über die Evolution würden die Feature-annotierten Modelle somit auf eine Art 175%-Modell erweitert werden. Um auf diesen ebenfalls gezielt Analysen ausführen zu können, muss auch hier eine Möglichkeit entwickelt werden, sie durch Slicing auf bestimmte Aspekte hin zu untersuchen.

Ziel der Arbeit

Zu diesem Zweck werden in dieser Masterarbeit zunächst die Grundlagen zu den Themen Softwareproduktlinien und Evolution - speziell Evolution von Softwareproduktlinien - sowie Feature-annotierte State Machines und Slicing erarbeitet. Basierend darauf wird dann ein Konzept entwor-

fen, welches es erlaubt, auch Evolution in annotierten State Machines darzustellen, sodass die 150%-Modelle auf 175%-Modelle ausgeweitet werden. Die 150%-Modell-Annotationen, welche bereits die Feature-Informationen beinhalten, könnten dabei ergänzt werden, sodass sie auf ähnliche Weise auch die Informationen darüber enthalten, für welche zeitliche Version der SPL das jeweilige Element gültig ist. Zudem soll die Slicing-Methode für 150%-Modelle erweitert werden, sodass diese auch für 175%-Modelle funktioniert. Dazu soll ein bestehendes Tool [21], welches bereits Feature-orientiertes Slicing von 150%-Modellen ermöglicht, erweitert werden, sodass es auch für 175%-Modelle gemäß des in dieser Arbeit entwickelten Darstellungskonzeptes und der Slicing-Methode funktioniert. Des Weiteren soll ein Algorithmus hergeleitet werden, welcher es ermöglicht, Higher-Order Deltas in 175%-Modelle umzuwandeln, damit die Betrachtung aus unterschiedlichen Perspektiven vereinfacht wird. Die Korrektheit des Algorithmus soll anschließend durch einen Beweis gezeigt werden. Weiterhin soll der Algorithmus als Tool implementiert werden. Mit Hilfe dessen können Evolutionsszenarien [29] zu vier delta-orientierten Fallstudien [11, 26] vom delta-orientierten Format in eine Form übertragen werden, die eine Verwendung für Feature-annotierte State Machines und deren Evolution erlaubt. Die Funktion des erweiterten Slicing-Tools wird anschließend mithilfe dieser Evolutionsszenarien evaluiert. Die Ergebnisse werden entsprechend in dieser Arbeit festgehalten und vorgestellt. Zusammenfassend werden in dieser Masterarbeit somit die folgenden Aufgaben bearbeitet:

- Einarbeitung in relevante Literatur zu den Themen Evolution von Softwareproduktlinien, Feature-annotierte State Machines und Slicing
- Entwicklung eines Konzeptes zur Modellierung von Evolution in Feature-annotierten State Machines
- Erweiterung eines bestehenden Tools zum Slicing von 150%-Modellen, sodass dieses auch für 175%-Modelle funktioniert
- Entwicklung eines Algorithmus zur Umwandlung von Higher-Order Deltas in 175%-Modelle sowie Implementierung dieses Algorithmus
- Evaluation der Funktion des Slicing-Tools und des Algorithmus
- Dokumentation der Ergebnisse

Strukturierung der Arbeit

In Kapitel 2 werden zunächst die Grundlagen für das Verständnis dieser Arbeit vermittelt. Dazu zählen etwa allgemeine Informationen über Softwareproduktlinien, Feature- und Delta-Modellierung (Kapitel 2.1) und die Erläuterung des Konzeptes von Feature-annotierten State Machines (Kapitel 2.2). Des Weiteren fällt darunter die Beschreibung der Funktionsweise von Slicing mit besonderem Fokus auf Modell-Slicing und Slicing von Feature-annotierten State Machines (Kapitel 2.3). Abschließend folgt eine ausführliche Betrachtung von Evolution in Softwareproduktlinien, bei der unter anderem verschiedene Möglichkeiten zur Modellierung von Evolution in Feature-Modellen und die Higher-Order Delta-Modellierung zur Modellierung von Evolution in Produktmodellen vorgestellt werden (Kapitel 2.4).

Danach folgt in Kapitel 3 die Entwicklung des Konzeptes zur Modellierung von Evolution in Feature-annotierten State Machines. Hierbei wird in Kapitel 3.1 zunächst eine geeignete Weise zur Darstellung von diesen sogenannten 175%-Modellen erarbeitet, auf deren Grundlage dann in Kapitel 3.2 eine formale Definition von 175%-Modellen gegeben wird. In Kapitel 3.3 wird das bereits für 150%-Modelle bestehende Slicing erweitert, sodass es ebenfalls auf 175%-Modelle angewendet werden kann.

Kapitel 4 beschäftigt sich mit der Konzeption des Algorithmus zur Umwandlung von Higher-Order Deltas in 175%-Modelle. Zu diesem Zweck werden in Kapitel 4.1 Regeln zur Repräsentation von Delta- und Element-Operationen als Annotationen definiert. Auf Basis dieser Regeln wird in Kapitel 4.2 der Transformationsalgorithmus konzipiert und beschrieben. In Kapitel 4.3 wird durch mathematische Induktion bewiesen, dass der Algorithmus ein äquivalentes 175%-Modell zum eingegebenen Higher-Order Delta-Modell erzeugt.

Daraufhin erfolgt in Kapitel 5 die Dokumentation der Implementierung des für 175%-Modelle erweiterten Slicing-Algorithmus und der Transformation von Higher-Order Delta- in 175%-Modelle. Dazu werden zunächst einige Aspekte der bereits bestehenden Implementierung beschrieben, um dann auf die Erweiterungen und neuen Funktionen einzugehen. Kapitel 6 widmet sich der Evaluation der Implementierung hinsichtlich des Zeitfaktors bei der Anwendung der Modell-Transformation und des 175%-Slicings. Abschließend folgt in Kapitel 7 eine Zusammenfassung der erarbeiteten Ergebnisse sowie ein Fazit über die gewonnenen Erkenntnisse als auch ein Ausblick für zukünftige Arbeiten.

2 Grundlagen

Dieses Kapitel behandelt die grundlegenden Informationen, die zum Verständnis dieser Arbeit nötig sind. Hierzu wird in Kapitel 2.1 zunächst das Konzept von *Softwareproduktlinien* [33] vorgestellt, sowie verschiedene Möglichkeiten Softwareproduktlinien zu modellieren. Aufbauend darauf werden in Kapitel 2.2 *Feature-annotierte State Machines* aus der Klasse der sogenannten *150%-Modelle* als Modellierungsmöglichkeit genauer betrachtet. Im Zuge dessen wird *Slicing* im Allgemeinen und speziell als Analysemethode für 150%-Modelle in Kapitel 2.3 beschrieben. Anschließend folgt in Kapitel 2.4 die Betrachtung von *Evolution* in Softwareproduktlinien als auch die Auseinandersetzung mit Konzepten zur Modellierung von Evolution.

2.1. Softwareproduktlinien

Bei einer *Softwareproduktlinie* handelt es sich um eine Reihe von ähnlichen, software-intensiven Produkten, die gemeinsame, unterschiedlich kombinierbare Merkmale - sogenannte *Features* - aufweisen und aus einem kollektiven Pool an Softwarebausteinen (engl. *assets*) auf eine vorher definierte Weise entwickelt werden [13]. Zu den gemeinsamen Softwarebausteinen zählen beispielsweise Anforderungen, Softwarearchitekturen oder Testfälle sowie komplette Komponenten oder Quellcode und unzählige weitere Artefakte des Softwareentwicklungsprozesses. Diese Softwarebausteine werden einmal implementiert und dann zur Erstellung so vieler Produkte der SPL wie möglich wiederverwendet.

Der Vorteil einer Softwareproduktlinie liegt darin, viele ähnliche, aber dennoch individuell unterschiedliche Produkte ohne erheblichen Mehraufwand anbieten zu können [33]. Auf diese Weise kann die Erfüllung individueller Kundenwünsche mit einer kostengünstigeren Massenproduktion (engl. *mass customization*) gekoppelt werden. Zu diesem Zweck sind Softwareproduktlinien explizit auf Wiederverwendung ausgerichtet und bieten größtmögliche Variabilität, um die individuelle Gestaltung von Produkten unter Ausnutzung der gemeinsamen Softwarebausteine zu gewährleisten. Neben der Reduzierung von Kosten kann auch eine kürzere Zeit bis zur Markteinführung eines Produktes erreicht werden. Dies ist möglich, da durch die bereits implementierten, wiederverwendbaren Artefakte der Entwicklungsprozess für das einzelne Produkt wesentlich verkürzt werden kann.

Pohl et al. [33] unterteilen den Entwicklungsprozess von Softwareproduktlinien und ihren Produkten in zwei verschiedene Phasen. Diese *Domain Engineering* und *Application Engineering* genannten Phasen, welche wiederum in weitere Teilphasen aufgegliedert sind, sind in der Pohl et al. [33] nachempfundenen Abbildung 2.1 dargestellt.

Domain Engineering, auch *Core Asset Development* [13], beschäftigt sich zum einen mit der Definition der Softwareproduktlinie durch genaues Abstecken des Geltungsbereiches und Identifizierung der Gemeinsamkeiten und Unterschiede der beinhalteten Produkte [33]. Zum anderen werden in dieser Phase direkt die wiederverwendbaren Bausteine der SPL implementiert, sodass diese später zur Erstellung der eigentlichen Produkte zur Verfügung stehen. Das Domain Engineering setzt sich zusammen aus den folgenden fünf Teilphasen [33]:

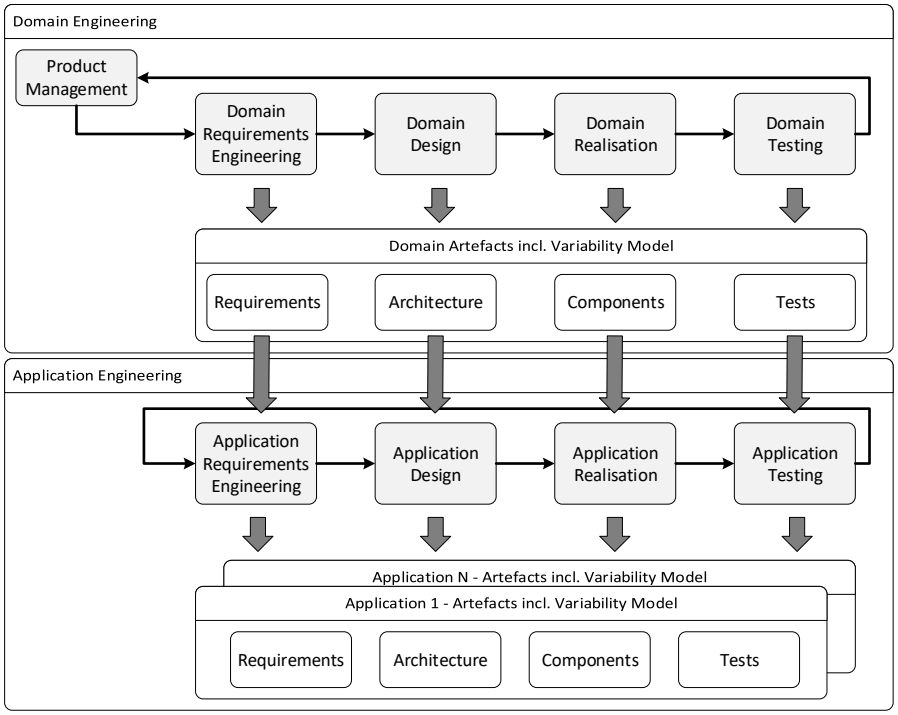


Abbildung 2.1.: Framework der Softwareproduktlinienentwicklung (nach Pohl [33])

1. **Product Management:** Beim Product Management wird zunächst der Geltungsbereich der Softwareproduktlinie bestimmt. Darüber hinaus werden die wesentlichen Features der Produktlinie ermittelt und in die Kategorien gemeinsame oder variable Features eingeordnet.

2. **Domain Requirements Engineering:** Diese Teilphase befasst sich mit der Erstellung von Variabilitätsmodell und wiederverwendbaren Anforderungen. Im Variabilitätsmodell ist die Variabilität der Softwareproduktlinie durch die Beschreibung der Variationspunkte und Varianten der SPL sowie deren Abhängigkeiten definiert.

3. **Domain Design:** Beim Domain Design wird die Referenzarchitektur für die Produktlinie gefertigt. Diese definiert eine Struktur, die für alle Produkte gültig ist, als auch Regeln, die bei der Erstellung von Produkten beachtet werden müssen.

4. **Domain Realisation:** In dieser Teilphase wird der genaue Entwurf der wiederverwendbaren Softwarekomponenten erstellt sowie deren Implementierung vorgenommen.

5. **Domain Testing:** Diese Phase beschäftigt sich mit der Erstellung von wiederverwendbaren Testartefakten wie etwa Testplänen oder Testfällen. Zusätzlich werden die bereits erstellten Softwarekomponenten auf die Erfüllung der Spezifikationen untersucht.

Application Engineering oder *Product Development* [13] ist die Phase, in der die konkreten Produkte der SPL erstellt werden [33]. Zu diesem Zweck werden so viele der gemeinsamen Bausteine aus dem Domain Engineering wie möglich wiederverwendet und unter Beachtung der Variabilität zu einem speziellen Produkt zusammengesetzt. Dabei soll so weit möglich das Implementieren von

produktspezifischen Bausteinen vermieden werden, da dies die Wartung und potenzielle Updates verkomplizieren würde. Application Engineering ist in die folgenden vier Teilphasen aufgeteilt [33]:

1. **Application Requirements Engineering:** Diese Teilphase beinhaltet die Erstellung der spezifischen Anforderungen für ein bestimmtes Produkt. Anhand dieser Anforderungen lassen sich sowohl die Teile des Produkts ableiten, welche bereits durch die gemeinsamen Bausteine der SPL implementiert sind, als auch Stellen identifizieren, an denen zusätzlich produktspezifische Lösungen umgesetzt werden müssen.

2. **Application Design:** Mithilfe der Referenzarchitektur wird in dieser Phase die Produktarchitektur erstellt, indem alle aus der Referenzarchitektur benötigten Teile übernommen und gegebenenfalls an spezifische Anforderungen des Produktes angepasst werden.

3. **Application Realisation:** Während der Application Realisation findet die Implementierung des konkreten Produkts unter Wiederverwendung von existierenden Softwarekomponenten statt. Hierbei kann unter Umständen ebenfalls eine spezifische Produktanpassung nötig sein.

4. **Application Testing:** In dieser letzten Teilphase erfolgt die Validierung und Verifikation des Produkts unter Verwendung von wiederverwendbaren als auch möglicherweise spezifisch erstellten Testartefakten.

Neben der Kostenreduzierung und der kürzeren Entwicklungszeit ergibt sich durch die Strukturierung des Entwicklungsprozesses auch eine gesteigerte Qualität der einzelnen Produkte [33]. Da der Entwicklungsprozess für jedes Produkt wiederverwendet wird, wirken sich nicht nur die Rückmeldungen für ein einzelnes Produkt sondern für alle Produkte positiv auf diesen aus. Gefundene Fehler in einem Produkt resultieren somit auch automatisch in Optimierungen für alle anderen Produkte, was die Qualität der kompletten Softwareproduktlinie insgesamt steigert.

Die grundlegende Basis, auf der Softwareproduktlinien aufbauen, ist daher ihre Variabilität [33]. Trotz der Vorteile, die sich ihretwegen ergeben, entstehen durch die Variabilität auch einige Schwierigkeiten, vor allem was die Modellierung von Softwareproduktlinien betrifft. Damit die Variabilität von Produktlinien ausreichend und in geeignetem Maße dargestellt werden kann, werden unterschiedliche Arten von Modellen verwendet, die im Folgenden vorgestellt werden.

Feature-Modellierung

Die von Kang et al. [22] eingeführten *Feature-Modelle* sind eine Möglichkeit zur Erfassung von Gemeinsamkeiten und Variabilität einer Softwareproduktlinie. Sie bestehen aus unterschiedlich variablen Features, welche hierarchisch strukturiert sind. Mithilfe eines Feature-Modells können die möglichen Produkte, auch *Produktvarianten* genannt, die sich aus einer SPL bilden lassen, abgeleitet werden. Dies ist möglich, indem die gültigen Feature-Konfigurationen des Feature-Modells betrachtet werden. Eine *Feature-Konfiguration* ist eine Auswahl an Features aus einem Feature-Modell. Gültig ist diese genau dann, wenn sie die vorgegebenen Beziehungen zwischen den einzelnen Features berücksichtigt. In diesem Fall beschreibt sie die Feature-Kombination für genau eine bestimmte Produktvariante der Softwareproduktlinie. Anhand einer gültigen Feature-Konfiguration können die Grundbausteine abgeleitet werden, die zur Erstellung der Produktvariante nötig sind. Diese Zusammenstellung von Grundbausteinen wird *Produktkonfiguration* genannt.

Grafisch lassen sich Feature-Modelle durch baumähnliche *Feature-Diagramme* wie etwa in Abbildung 2.2 darstellen [5]. Die Features bilden hierin die Knoten des Feature-Diagramms. Kind-Features dienen dabei zum weiteren Spezifizieren ihres übergeordneten Eltern-Features. Jedes Feature nimmt eine Variabilitätseigenschaft an, welche beschreibt, auf welche Weise das Feature in der SPL variabel ist:

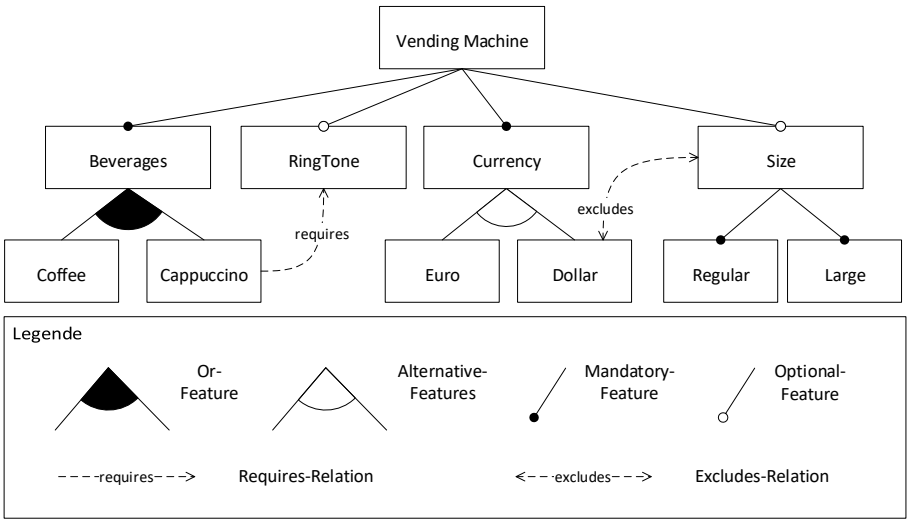


Abbildung 2.2.: Ein beispielhaftes Feature-Diagramm

- **Mandatory-Features:** Diese Features beschreiben die Gemeinsamkeiten der Produktlinie. Sie sind somit in jedem Produkt der SPL enthalten und sind daher auch in jeder gültigen Feature-Konfiguration fest verankert.
- **Optional-Features:** Bei diesen Features handelt es sich um die variablen Teile der SPL. Es ist somit dem Kunden überlassen, diese Features seiner gewünschten Feature-Konfiguration hinzuzufügen oder nicht.

Die Kennzeichnung der Mandatory-Feature in Feature-Diagrammen wird durch einen schwarz ausgefüllten Kreis oberhalb des Features vorgenommen, während die Optional-Features durch einen nicht beziehungsweise weiß ausgefüllten Kreis gekennzeichnet werden [5]. In Abbildung 2.2 handelt es sich somit bei den Features `Beverages` und `Currency` um Mandatory-Features sowie bei `RingTone` und `Size` um Optional-Features. Beim Wurzel-Feature eines Feature-Diagramms handelt es sich zwar um ein obligatorisches Feature, das ebenfalls für jede gültige Feature-Konfiguration selektiert werden muss, jedoch wird dieses nicht als Mandatory-Feature gekennzeichnet.

Zusätzlich zu den einfachen Variabilitätseigenschaften *mandatory* und *optional* existieren noch komplexere Zusammenhänge für die Variabilität von Kindknoten, welche die Möglichkeiten bei deren Auswahl einschränken können [5]:

- **Alternative-Features:** Für Alternative-Features gilt, dass aus einer beliebigen Anzahl von Kind-Features, die auf diese Weise verbunden sind, nur genau ein Feature für eine gültige Feature-Konfiguration ausgewählt werden darf.

- **Or-Features:** Aus einer zusammenhängenden Gruppe von Or-Features muss mindestens ein Feature ausgewählt werden. Zusätzlich können beliebig viele weitere bis alle Features der Gruppe selektiert werden.

Vorraussetzung für die Selektierung von Or- oder Alternative-Features ist, dass das jeweilige Eltern-Feature ebenfalls für die Feature-Konfiguration ausgewählt wurde [5]. Or-Features werden in einem Feature-Diagramm durch einen schwarz ausgefüllten Winkel unterhalb des Eltern-Features gekennzeichnet, während Alternative-Features durch einen nicht beziehungsweise weiß ausgefüllten Winkel verbunden sind. Die Features `Euro` und `Dollar` in Abbildung 2.2 sind somit Alternativ-Features, während es sich bei `Coffee` und `Cappuccino` um Or-Features handelt.

Neben den bereits genannten Variabilitätsmöglichkeiten können verschiedene Features auch ebenenübergreifend miteinander in Beziehung stehen [5]:

- **Requires-Relation:** Diese Relation definiert, welche Features für eine gültige Feature-Konfiguration zwingend gewählt werden müssen, wenn das Ausgangsfeature selektiert wurde.
- **Exludes-Relation:** Bei Features, die durch eine Excludes-Relation verbunden sind, handelt es sich um Features, die sich gegenseitig ausschließen. Eine gemeinsame Selektion in einer gültigen Feature-Konfiguration ist daher nicht möglich.

Diese Relationen werden auch *Cross-Tree Constraints* genannt und können in Feature-Diagrammen entweder durch aussagenlogische Formeln oder wie in Abbildung 2.2 durch gestrichelte Linien mit den Labeln `requires` beziehungsweise `excludes` repräsentiert werden [5]. Eine Requires-Relation ist dabei einfach gerichtet, während eine Excludes-Relation doppelt gerichtet ist.

Feature-Modelle sind somit eine geeignete Möglichkeit, um eine Übersicht über die gesamte Softwareproduktlinie und ihre Varianten zu erhalten. Die Betrachtung von Variabilität in den Produkten selbst, das heißt auf Artefaktlevel, kann damit jedoch nicht vorgenommen werden.

Variabilitätsmodellierung von Entwicklungsartefakten

Damit die Variabilität der Softwareproduktlinie auch in beispielsweise Komponentendiagrammen oder State Machines dargestellt werden kann, ist daher eine andere Art der Modellierung erforderlich [35]. Diese Darstellungen können mitunter sehr komplex ausfallen, da etwa voneinander abhängige oder sich ausschließende Features komplizierte Auswirkungen auf das resultierende Produktmodell haben können. Um diese Sachverhalte dennoch möglichst einfach und verständlich abbilden zu können, existieren unterschiedliche Ansätze zur Modellierung von Variabilität auf Artefaktebene. Diese Ansätze werden dabei in die folgenden drei verschiedenen Kategorien eingeteilt [35]:

- **Annotativ:** Annotative Ansätze kapseln die Inhalte für sämtliche Produkte einer Produktlinie in einem großen Produktmodell, auch bezeichnet als 150%-Modell. Die Inhalte sind mit Informationen darüber annotiert, auf welches Produkt sie sich beziehen. Da dieses Modell auch sich gegenseitig ausschließende Inhalte enthalten kann, handelt es sich meist nicht um ein funktionsfähiges Produktmodell. Indem jegliche Inhalte des Modells entfernt werden, die nicht für ein bestimmtes Produkt benötigt werden, kann das entsprechende Produktmodell gebildet werden.

- **Kompositional:** Bei kompositionalen Ansätzen existieren viele einzelne Modellfragmente, welche jeweils für bestimmte Produkte gültig sind. Um ein Produktmodell für ein bestimmtes Produkt zu erhalten, wird das Modell aus genau jenen Modellfragmenten zusammengesetzt, die sich auf dieses Produkt beziehen.
- **Transformational:** Transformationale Ansätze bestehen aus einem Kernmodell, welches jene Modellinhalte enthält, die für alle Produkte gültig sind, und aus einer Menge von Regeln oder Operationen. Letztere definieren, welche Änderungen am Kernmodell vorgenommen werden müssen, damit ein bestimmtes Produktmodell daraus resultiert.

Für die verschiedenen Kategorien der Modellierungsansätze ergeben sich unterschiedliche Vor- und Nachteile [35]. Annotative Ansätze zeichnen sich dadurch aus, dass sie eine Übersicht über sämtliche Inhalte der verschiedenen Produktvarianten bieten und dabei eine detaillierte Darstellung der Unterschiede zwischen den Varianten gestatten. Nachteilig wirkt sich hierbei jedoch aus, dass die Modelle durch die Fülle an Informationen und aufgrund fehlender Modularität sehr groß werden können, was wiederum zu Unübersichtlichkeit führt. Kompositionale Ansätze hingegen bieten durch die kleinen Modellfragmente eine sehr hohe Modularität und somit auch Übersichtlichkeit. Dies geht allerdings zu Lasten eines fehlenden Gesamtüberblicks als auch teilweise mangelnder Flexibilität. So fehlt es den kompositionalen Ansätzen beispielsweise an Aussagekraft, was das Entfernen von Produktverhalten durch Auswahl von bestimmten Features betrifft. Durch transformationale Ansätze werden die Vorteile von annotativen und kompositionalen Ansätzen in einem Ansatz vereint, da sie sowohl Modularität als auch Flexibilität für komplexe Sachverhalte bieten. Allerdings fehlt auch hier wieder ein Gesamtüberblick über alle Möglichkeiten, da Produktmodelle nur durch Kombinationen von Kernmodell und Operationen entstehen.

Da alle drei Kategorien von Ansätzen unterschiedliche Vor- und Nachteile bieten, gibt es keine mustergültige Lösung zur Darstellung von Variabilität in Softwareproduktlinien auf Artefaktebene [35]. Welcher Ansatz gewählt wird, hängt hauptsächlich davon ab, welche Ziele verfolgt werden und unter welchem Aspekt die Produktlinie betrachtet werden soll. Unter Umständen kann es nötig sein, verschiedene Ansätze zu kombinieren, um alle gewünschten Aspekte abzudecken.

In dieser Arbeit liegt der Fokus auf Feature-annotierten State Machines [21], da diese gleichzeitig einen Gesamtüberblick über alle möglichen Modellinhalte sowie detaillierte Informationen über deren Anwendungsbedingungen erlauben. Feature-annotierte State-Machines sind ein annotativer Ansatz und werden in Kapitel 2.2 ausführlich vorgestellt. Darüber hinaus wird im Verlauf der Arbeit mit Delta-Modellierung [10] ein transformationaler Ansatz genutzt. Dieser dient als Basis für Higher-Order Delta-Modelle (s. Kapitel 2.4), welche das Ausgangsmodell des zu entwickelnden Transformationsalgorithmus bilden. Delta-Modellierung findet hierbei Anwendung, da die zur Evaluation verwendeten Fallstudien [11, 26] und Evolutionsszenarien [29] bereits delta-orientiert vorliegen. Aus diesem Grund wird im Folgenden auch Delta-Modellierung kurz vorgestellt.

Delta-Modellierung

Delta-Modellierung [10] ist ein transformationaler Ansatz zur Modellierung von Variabilität in Softwareproduktlinien, der unabhängig von Modellier- oder Programmiersprachen ist und auf jegliche Art von Produktmodell $M = (O, R)$ angewandt werden kann [35]. Dabei ist $O \subset \mathcal{O}$ eine Menge von Modellobjekten und $R \subseteq O \times O \subset \mathcal{R}$ eine modell-spezifische Relation zwischen den Modellobjekten. $\mathcal{O}$ bezeichnet an dieser Stelle das Universum aller möglichen Objekte für den entsprechenden

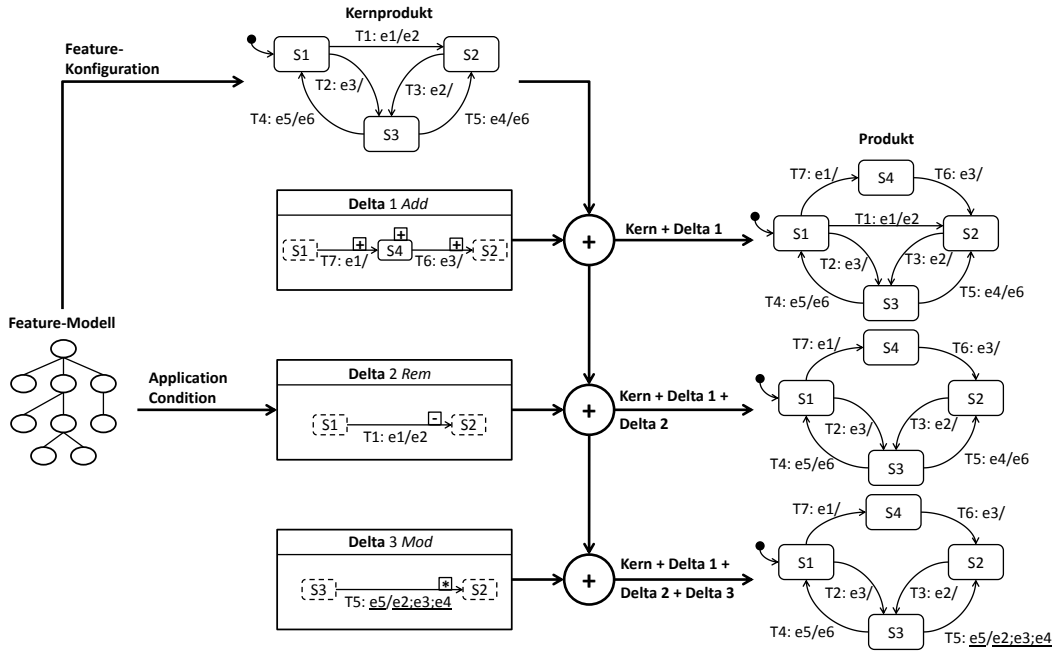


Abbildung 2.3.: Delta-Konzept am Beispiel einer State Machine [24]

Modelltyp. Diese Objekte sind etwa Zustände für State Machines oder Klassen und Schnittstellen für Klassendiagramme. $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{O}$ bezeichnet folglich die Relation über das Universum der Modellobjekte. Für State Machines ist die Relation repräsentiert durch Transitionen, während sie in Klassendiagrammen etwa durch Kompositionen oder Assoziationen verkörpert werden kann.

Ein *Delta-Modell* $DM = (M_{core}, \Delta)$ repräsentiert die verschiedenen Produktvarianten einer Softwareproduktlinie. Es setzt sich zusammen aus einem Kernmodell $M_{core} = (O, R)$ sowie einer Menge von Deltas $\Delta = \{\delta_1, ..., \delta_n\}$. Das *Kernmodell* oder auch Kernprodukt ist hierbei ein gültiges Produkt der Softwareproduktlinie, bestehend aus Features, die möglichst vielen Produkten der SPL gemein sind. Welches Produkt sich genau als Kernmodell eignet, ist hierbei von Produktlinie zu Produktlinie verschieden, meist ist es jedoch ein vorzugsweise einfaches Standardprodukt. Ein *Delta* $\delta = \{op_1, ..., op_n\}$ ist eine Zusammensetzung aus Operationen $op = (\text{add } e / \text{rem } e / \text{mod } (e, e'))$ mit $e \in O \cup R$, die beschreiben, welche Änderungen am Kernmodell erfolgen müssen, damit ein gewünschtes Produktmodell als Ergebnis resultiert. Als Operationen gelten das Hinzufügen (*Add*), Entfernen (*Remove*) oder Modifizieren (*Modify*) von Modellinhalten. Modellinhalte können hierbei die Modellelemente der verschiedenen Modellformen wie beispielsweise Komponenten, Klassen oder Zustände sein als auch Relationen zwischen diesen Elementen. In einem Delta kann eine beliebige Anzahl an Operationen gekapselt werden.

Die Herleitung eines spezifischen Produktmodells aus dem Kernmodell erfolgt, indem die in den Deltas beinhalteten Operationen auf das Kernmodell angewendet werden und dieses somit verändern [35]. Die Deltas sind hierbei den unterschiedlichen Produkten durch eine *Anwendungsbedingung* (engl. *application condition*) zugeordnet. Die Anwendungsbedingung kann theoretisch für unterschiedliche Kriterien verwendet werden, meist spezifiziert sie jedoch für welches Feature ein Delta gilt. Hierbei ist nicht jedem Delta genau ein Feature zugeordnet, sondern ein Delta kann auch

sowohl für eine Deselektion von Features als auch für eine bestimmte Kombination aus Features bis hin zu teilweisen oder vollständigen Feature-Konfigurationen gelten. Daraus ergibt sich, dass häufig mehrere, unterschiedliche Deltas sukzessiv auf das Kernmodell angewendet werden müssen, um das Produktmodell für eine bestimmte Feature-Konfiguration zu erhalten.

Damit bei dieser aufeinanderfolgenden Anwendung von Deltas keine Konflikte auftreten, unterliegen die Deltas außerdem einer Ordnungsrelation. Diese definiert, welche Deltas zuvor angewendet werden müssen, bevor ein bestimmtes Delta benutzt werden darf. Dadurch wird verhindert, dass beispielsweise versucht wird, Inhalte einzufügen, die schon im Modell enthalten sind, oder umgekehrt Inhalte zu entfernen, obwohl sie nicht im Modell existieren. Ebenso wird so garantiert, dass Inhalte, die modifiziert werden sollen, bereits im Modell enthalten sind. Auf diese Weise soll bei der Delta-Modellierung durch die Ordnungsrelation in jedem Schritt die Wohlgeformtheit und Konfliktfreiheit von Modellen sichergestellt werden.

Abbildung 2.3 zeigt eine schematische Darstellung für die Anwendung von Deltas auf ein Kernmodell. Für eine gültige Feature-Konfiguration aus dem Feature-Modell werden die Deltas ausgewählt, deren Anwendungsbedingungen der Konfiguration entprechen. Diese werden dann schrittweise nacheinander auf das Kernmodell angewendet. Nach jedem Schritt ergibt sich ein neues Zwischenmodell bis das gewünschte Produktmodell erreicht ist. Die Zwischenmodelle können wiederum Produktmodelle für andere Feature-Konfigurationen sein.

Nachdem nun Delta-Modellierung als transformationaler Ansatz zur Modellierung von Variabilität vorgestellt wurde, werden im Folgenden als annotatives Konzept Feature-annotierte State Machines näher erläutert.

2.2. Feature-annotierte State Machines

Feature-annotierte State Machines (FaSMs) sind ein annotativer Ansatz, mit dem die Variabilität in Softwareproduktlinien speziell für Zustandsautomaten modelliert werden kann [21]. Während Annotationen prinzipiell für jede Art von Produktmodell verwendet werden können, beschränken sich FaSMs somit auf die Verhaltensspezifikation eines Systems in Form von State Machines.

Die Beschreibung des Verhaltens wird bei State Machines durch eine Menge von Zuständen S und eine Menge von Transitionen T zwischen den Zuständen beschrieben [21]. Die Transitionen $t = (s, l, s') \in T$ besitzen ein Label (Transitionsbeschriftung) l aus einer Menge von Labeln L der Form $ev[g]/a$. Die Label beschreiben, welches Verhalten beim Übergang vom Ausgangszustand $s \in S$ zum Zielzustand $s' \in S$ der Transition gezeigt wird. ev beschreibt das Event, bei dessen Auftreten der Übergang ausgelöst wird, g bezeichnet den Guard - eine Bedingung, die erfüllt sein muss, damit der Übergang erfolgen kann - und A kennzeichnet die Aktionen, die bei Übergang ausgeführt werden. Eine Aktion kann etwa das Auslösen eines Events oder die Wertzuweisung einer Variable sein. Es existiert keine Beschränkung der Anzahl an Aktionen, welche pro Übergang ausgeführt werden können. Alle drei Teile des Labels sind dabei optional, müssen also nicht zwangsläufig definiert werden. Ein leeres Label ist daher ebenso möglich. Die Menge von Verhaltensweisen, die durch eine State Machine spezifiziert werden, wird in Form einer Menge von Sequenzen von Transitionen $t_1 t_2 ... t_i$, sogenannten Pfaden, repräsentiert.

Abbildung 2.4 zeigt eine stark vereinfachte State Machine eines Verkaufsautomaten. Bei den Transitionen `t1` bis `t3` wird als Event eines der Getränke Kaffee, Tee oder Cappuccino ausgewählt. Diese Auswahl wird von der ausgelösten Aktion in der Variable `drink` gespeichert. Bei den Transitionen

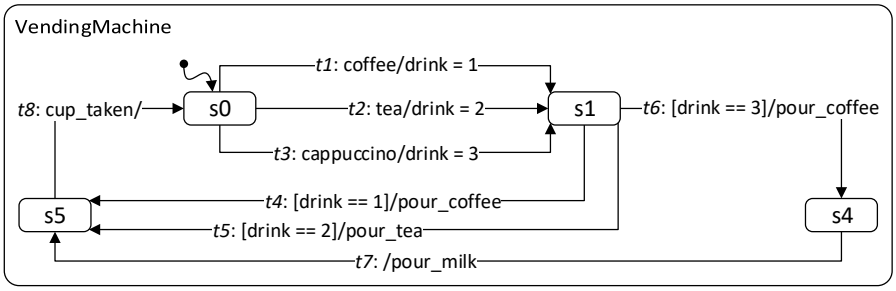


Abbildung 2.4.: State Machine eines Verkaufsautomaten

t4 bis t6 erfolgt die Auswahl des nächsten Übergangs ohne Event durch die Prüfung des Guards. Je nach gespeichertem Wert der Variable drink wird das entsprechende Getränk eingegossen. Bei t7 wird ohne weiteres Event Milch für den Cappuccino hinzugefügt und bei t8 wird durch das Entnehmen des Bechers der Übergang ohne das Ausführen einer Aktion ausgelöst.

Da sie als 150%-Modell sämtliche Inhalte aller State Machines der Softwareproduktlinie in sich vereinen, bilden Feature-annotierte State Machines meist keine funktionsfähigen Produktmodelle [21]. Damit die Modellelemente von 150%-Modellen den verschiedenen Produkten zugeordnet werden können, sind sie mit Selektionsbedingungen annotiert. In vielen Fällen handelt es sich bei diesen Bedingungen um Features, da sich Produktvarianten leicht durch Feature-Konfigurationen definieren lassen. Feature-annotierte State Machines werden daher in Kombination mit einem Feature-Modell verwendet, auf welches sich die Selektionsbedingungen beziehen. Kamischke et al. [21] nutzen die Repräsentation eines Feature-Modells $FM \in \mathbb{B}(F)$ als aussagenlogische Formel über Feature-Parameter von Batory [4]. Die Features des Feature-Modells sind definiert als Menge $F = \{f_1, f_2, ..., f_n\}$. $\mathbb{B}(F)$ bezeichnet die Menge aller möglichen aussagenlogischen Formeln, die durch die Menge F gebildet werden können.

Eine *Annotationsfunktion* $\alpha : E \to \mathbb{B}(F)$ ordnet jedem Modellinhalt $e \in E$ der Feature-annotierten State Machine eine *Selektionsbedingung* $\alpha(e)$ zu, wobei E die Menge aller Zustände und Transitionen des Modells ist. Die Selektionsbedingungen existieren dabei in Form von aussagenlogischen Formeln über Feature-Parametern. Hierbei muss $\alpha(e) \vDash FM$ gelten. Die Selektionsbedingung muss also die gegebenen Einschränkungen des Feature-Modells erfüllen. Eine Feature-annotierte State Machine (M, α) besteht somit aus einem State Machine Modell M, dessen Elemente $e \in E$ mit Selektionsbedingungen $\alpha(e)$ annotiert sind.

Die FaSM in Abbildung 2.5 ist mit einzelnen Features annotiert. Die Transitionen t3 und t4 sind nicht annotiert und daher immer im Modell enthalten. Transition t1 taucht nur dann im Produktmodell auf, wenn das Feature Dollar selektiert wurde. Im Gegenzug ist Transition t19 immer genau dann vorhanden, wenn das Feature RingTone *nicht* selektiert wurde.

Mit über mehr als einem Feature-Parameter gebildeten Formeln als Selektionsbedingung kann die Auswahl eines Elements noch weiter beschränkt werden [21]. Auf diese Weise kann etwa definiert werden, dass ein Element nur dann im Produktmodell vorhanden ist, wenn sowohl Kaffee als auch Cappuccino selektiert wurden, oder wenn Euro selektiert wurde, aber Tee nicht. Ein bestimmtes Produktmodell für eine Produktvariante kann aus einem 150%-Modell abgeleitet werden,

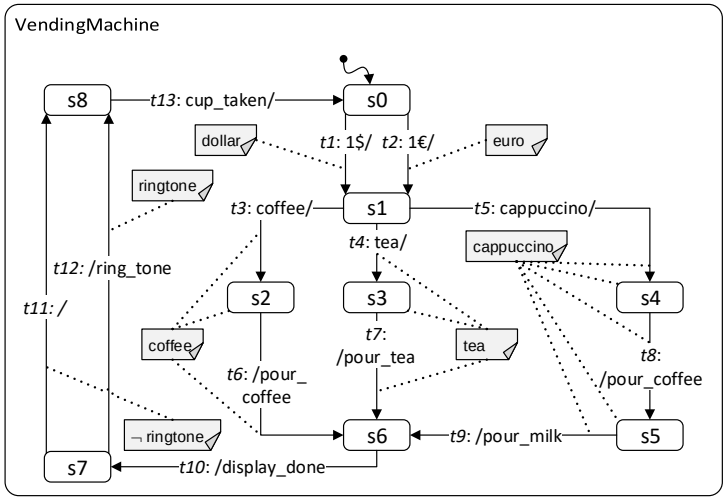


Abbildung 2.5.: Feature-annotierte State Machine eines Verkaufsautomaten

indem all diejenigen Elemente für das Produktmodell ausgewählt werden, deren Selektionsbedingung $\alpha(e)$ der Feature-Konfiguration Γ für die Produktvariante entspricht. Γ ist dabei definiert als Funktion $\Gamma : F \rightarrow \mathbb{B}$ mit $\mathbb{B} = \{false, true\}$. Jedem Feature $f \in F$ wird somit ein boolscher Wert zugeordnet. $\Gamma(f) = true$ bezeichnet dabei ein in der Konfiguration selektiertes Feature, während $\Gamma(f) = false$ ein unselektiertes Feature charakterisiert. Für eine gültige Feature-Konfiguration Γ muss dabei gelten, dass $\Gamma \vDash FM$. Γ kann hierbei sowohl eine partielle als auch eine vollständige Konfiguration sein. Um eine *partielle Feature-Konfiguration* handelt es sich, wenn Γ eine partielle Funktion über F bildet, das heißt, wenn nicht jedem $f \in F$ ein Wert zugeordnet ist. Alle gültigen - sowohl partiellen als auch vollständigen - Feature-Konfigurationen sind zusammengefasst im gültigen Produktraum PC_{FM}. Für eine gültige Konfiguration $\Gamma \in PC_{FM}$ wird ein Element $e \in E$ aus der Feature-annotierten State Machine also genau dann in die State Machine für Γ gewählt, wenn gilt $\Gamma \vDash \alpha(e)$. Bei Features, die Kernfunktionen repräsentieren, kann die Annotation weggelassen werden, da diese mit *true* annotiert werden.

Damit die Wohlgeformtheit der ableitbaren Produktmodelle garantiert werden kann, müssen Feature-annotierte State Machines den folgenden Kriterien für Wohlgeformtheit entsprechen [21]:

- Für jeden Zustand $s \in S$, der eingebettet ist in einen übergeordneten Zustand s', gilt, dass $\alpha(s) \Rightarrow \alpha(s')$. Die Anwesenheit eines Zustandes bewirkt somit die Anwesenheit aller übergeordneten Zustände bis zum Wurzelzustand.
- Für jede Transition $t = (s, l, s') \in T$ gilt, dass $\alpha(t) \Rightarrow \alpha(s)$ und $\alpha(t) \Rightarrow \alpha(s')$. Die Anwesenheit einer Transition bewirkt somit die Anwesenheit ihres Ausgangs- und Zielzustands.
- Jeder Zustand $s_k \in S$ muss mindestens über einen Pfad $t_1 t_2 ... t_k$ erreichbar sein, sodass gilt: $\alpha(s) \Rightarrow \alpha(t_i)$ für $1 \leq i \leq k$.

Feature-annotierte State Machines bieten also insgesamt einen detaillierten Überblick über das Verhalten sämtlicher Produkte einer Softwareproduktlinie [21]. Dabei beschreiben sie durch die Feature-Annotationen genau, welches gezeigte Verhalten für welches Produkt gilt. Diese Modelle

können beispielsweise zu Analysezwecken oder auch zur Testfallgenerierung [8, 9] verwendet werden. Nachteil ist hierbei, dass FaSMs sehr groß werden können, was sowohl die Übersichtlichkeit als auch die Analyse der Modelle ziemlich erschwert. Daher ist es nützlich, die 150%-Modelle auf die Elemente des Modells reduzieren zu können, die für ein bestimmtes Betrachtungskriterium wichtig sind. So kann etwa beim modellbasierten Testen eine gezielte Testfallgenerierung für ein bestimmtes Teilmodell erfolgen, wenn Testziele als Kriterium gewählt werden und somit nur noch die für das Testziel relevanten Modellinhalte zur Generierung herangezogen werden. Um diese Reduktion zu erreichen, kann Modell-Slicing verwendet werden. Im folgenden Kapitel wird daher Slicing zuerst im Allgemeinen und dann speziell für Modelle näher vorgestellt.

2.3. Slicing

Das zuerst von Weiser [39] vorgestellte *Slicing* stammt ursprünglich aus der statischen Programmanalyse und dient dazu, Programmcode auf die für ein bestimmtes Analyseziel relevanten Teile zu reduzieren [38]. Dieses Vorgehen verringert beispielsweise den Aufwand bei der Suche nach Fehlern im Programmcode. Ergibt sich etwa für eine Variable nicht der gewünschte Ausgabewert, kann das Programm auf die Anweisungen getrimmt werden, die zur Berechnung des Wertes beitragen. Mithilfe von Slicing kann somit eine effizientere Fehleranalyse erfolgen, da irrelevante Programmteile bereits herausgefiltert wurden und betroffene Stellen so schneller ausfindig gemacht werden können. Zusätzlich kann Slicing zum Beispiel auch zu einem besseren Programmverständnis beitragen oder dabei helfen, bestimmte Softwaremetriken, wie etwa die Kohäsion eines Programmes, zu messen.

Beim Slicing wird dazu anhand eines *Slicing-Kriteriums* C aus einem Programm P ein sogenannter *Slice* S erzeugt [38]. Ein Slice ist dabei eine Untermenge an Anweisungen aus P. Diese Untermenge bildet wiederum selbst ein ausführbares Programm und muss dabei exakt dasselbe Verhalten zeigen wie zuvor in P [39]. Das Slicing-Kriterium setzt sich zusammen aus einer Untermenge V an Variablen aus Programm P und einer bestimmten Programmanweisung n. Das Kriterium ist somit definiert als $C = (n, V)$. Unter Anwendung von $C = (n, V)$ ergeben sich als Slice all die Anweisungen, welche die Variablen V an Programmpunkt n beeinflussen [38].

Es existieren unterschiedliche Varianten des Slicings, die sich durch verschiedene Eigenschaften unterscheiden [38]. Slicing kann zum einen unterschieden werden anhand der Richtung aus der die Betrachtung erfolgt [16]:

- **Backward-Slicing:** Beim Backward-Slicing enthält der Slice all diejenigen Anweisungen, die das gewählte Slicing-Kriterium auf irgendeine Art und Weise beeinflussen. Dies kann direkt, etwa durch eine Defintion, oder indirekt, wie beispielweise einen Schleifenaufruf geschehen. Das ursprüunglich von Weiser [39] propagierte Slicing fällt in diese Kategorie.
- **Forward-Slicing:** Beim Forward-Slicing werden für den Slice die Anweisungen berechnet, die vom gewählten Kriterium beeinflusst werden. Dies können also beispielsweise Anweisungen sein, welche die im Kriterium enthaltenen Variablen für Berechnungen verwenden.

Des Weiteren lässt sich Slicing in statisches und dynamisches Slicing unterteilen [38]:

- **Statisches Slicing:** Beim statischen Slicing wird der Slice nur aus den Informationen berechnet, die zur Übersetzungszeit des Programms verfügbar sind. Der sich ergebende Slice enthält

somit alle Anweisungen, die das Kriterium unabhängig von konkreten Eingabewerten beeinflussen könnten.

- **Dynamisches Slicing:** Beim dynamischen Slicing wird der Slice für eine bestimmte Ausführung des Programms, somit also auch für bestimmte Eingabewerte ermittelt. Das Slicing-Kriterium enthält daher zusätzlich zu n und V konkrete Eingabewerte für das Programm. Auf diese Weise ist es möglich, kleinere und somit genauere Slices zu erhalten. Da diese allerdings von bestimmten Eingabewerten abhängig sind, sind sie nicht mehr allgemeingültig.

Für verschiedene Slicing-Methoden mit unterschiedlichen Eigenschaften ergeben sich somit auch unterschiedliche Slices S bei gleichem C für ein Programm P [38]. P ist dabei immer auch selbst ein Slice von P. Auf diese Weise können durch unterschiedliche Methoden je nach Zielsetzung passende Slices erzeugt werden. Bei der Fehleranalyse etwa sind möglichst minimale Slices erwünscht, um den Aufwand bei der Suche so weit wie möglich zu reduzieren.

Das statische Slicing ist dabei das am meisten verbreitete Slicing [38]. Ein Slice wird hierbei erstellt, indem zuerst rekursiv alle direkt relevanten Variablen $R_C^0(i)$ für alle Anweisungen i bestimmt werden. Dazu werden die Mengen $REF(i)$ aller in Anweisung i referenzierten und $DEF(i)$ aller in Anweisung i definierten Variablen benötigt. Eine Variable v gehört genau dann zu den direkt relevanten Variablen, wenn gilt:

- $v \in V$ und $i = n$
  Das bedeutet, dass jede im Kriterium spezifizierte Variable der im Kriterium spezifizierten Anweisung eine direkt relevante Variable ist.

- i besitzt einen beliebigen Nachfolger j und
  - $v \in REF(i)$ und $DEF(i) \cap R_C^0(j) \neq \varnothing$
    v ist somit eine direkt relevante Variable, wenn v verwendet wird, um eine Variable zu definieren, die zu den direkt relevanten Variablen eines Nachfolgers von i gehört.
  - $v \in R_C^0(j)$ und $v \notin DEF(i)$
    v ist eine direkt relevante Variable, wenn v zu den direkt relevanten Variablen eines Nachfolgers von i gehört und nicht in i neu definiert wird.

Anhand der direkt relevanten Variablen werden die *direkt relevanten Anweisungen* S_C^0 bestimmt [38]. Dies sind alle Anweisungen, welche eine Variable definieren, die zu den direkt relevanten Variablen einer nachfolgenden Anweisung gehören. Hinzu kommen die Anweisungen B_C^k, welche auf irgendeine Weise die Ausführung der direkt relevanten Anweisungen kontrollieren, beispielsweise in Form von Schleifen oder If-Anweisungen. In einer weiteren Iteration folgen die *indirekt relevanten Variablen* $R_C^{(k+1)}(i)$. Dies sind alle Variablen, die für die *Kontrollanweisungen* B_C^k benötigt werden. Für die indirekt relevanten Variablen folgen wiederum die *indirekt relevanten Anweisungen* $S_C^{(k+1)}$, welche diese Variablen definieren. Zu $S_C^{(k+1)}$ gehören auch die Kontrollanweisungen. Auf die geschilderte Weise ergeben sich weitere Iterationen bis ein Fixpunkt erreicht ist. Dieser Fixpunkt bildet den gesuchten Slice.

Algorithmus 2.1 zeigt ein Beispielprogramm P, welches für eine Eingabe m sowohl die Summe als auch das Produkt aller ganzen Zahlen $\leq m$ berechnet. Algorithmus 2.2 zeigt den Slice, der sich

aus P für das Kriterium $C = (10, sum)$ ergibt. Hierbei wurden alle Anweisungen, die keine Auswirkung auf die Variable *sum* in Zeile 10 haben, nämlich die Zeilen 4, 7 und 9, entfernt. *sum* und *i* gehören hierbei zu den direkt relevanten Variablen $R_C^0(i)$, die Anweisungen 2, 3, 6, 8 und 10 zu den direkt relevanten Anweisungen S_C^0, Anweisung 5 zu den Kontrollanweisungen B_C^k, *n* zu den indirekt relevanten Variablen $R_C^{(k+1)}(i)$ und Anweisung 1 zu den indirekt relevanten Anweisungen $S_C^{(k+1)}$.

Algorithmus 2.1 .: Beispielprogramm P

```
1  read(m);
2  i := 1;
3  sum := 0;
4  product := 1;
5  while i <= m do
6  |  sum := sum + i;
7  |  product := product * i;
8  |  i := i + 1;
9  write(product);
10 write(sum);
```

Algorithmus 2.2 .: Slice von P für $C = (10, sum)$

```
1  read(m);
2  i := 1;
3  sum := 0;
4
5  while i <= m do
6  |  sum := sum + i;
7  |
8  |  i := i + 1;
9
10 write(sum);
```

Als Erweiterung des bisher beschriebenen Slicings existiert außerdem das *bedingte Slicing* (engl. *conditioned slicing*) [16]. Dieses zieht eine Brücke zwischen statischem und dynamischem Slicing. Beim bedingten Slicing wird C um eine Bedingung ϕ erweitert [21]. Ein bedingter Slice enthält demnach nur alle diejenigen Anweisungen, die sowohl dem einfachen Kriterium (n, V) entsprechen als auch ϕ erfüllen.

Algorithmus 2.3 .: Beispielprogramm Q

```
1  read(a);
2  if a < 0 then
3  |  a := -a;
4  x := 1/a;
```

Algorithmus 2.4 .: Bedingter Slice von Q für $C = (4, a, a > 0)$

```
1 read(a);
2
3
4 x := 1/a;
```

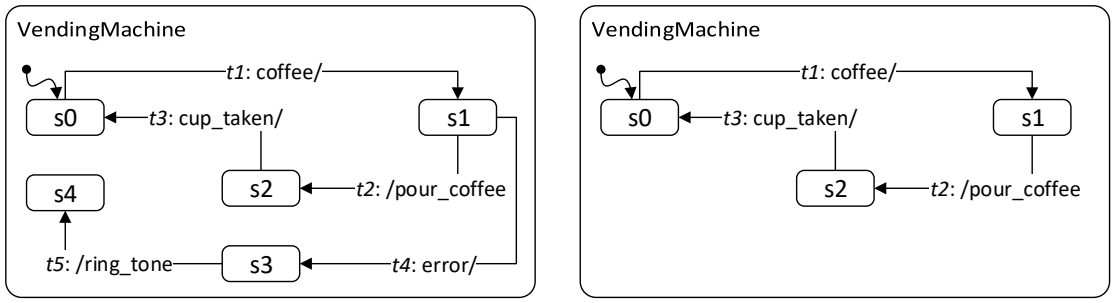
Algorithmus 2.3 zeigt ein Programm, dass eine Zahl a einliest und dann, sollte die Zahl positiv sein, 1 durch diese Zahl teilt. Ist a negativ, wird die Zahl zuvor in eine positive Zahl umgewandelt. Für ein Kriterium $(4, a)$ würde sich als normaler Slice wieder Q selbst ergeben. Betrachtet man allerdings ein bedingtes Kriterium $(4, a, a > 0)$, dann ergibt sich als Slice Algorithmus 2.4 [38].

Slicing kann neben der Programmanalyse außerdem adaptiert werden, um zur Analyse von Modellen verwendet zu werden [21]. Dieses sogenannte Modell-Slicing wird im Folgenden beschrieben.

Modell-Slicing

Für sehr große Modelle, wie beispielsweise Feature-annotierte State Machines, ergeben sich bezüglich ihrer Komplexität häufig Probleme, was eine effiziente Analyse betrifft [21]. So kann sich etwa modellbasiertes Testen oder Model Checking zur automatischen Prüfung der Korrektheit von Modellen aufgrund der Menge an Zuständen als schwierig erweisen. Modell-Slicing [3] bietet daher eine Möglichkeit, Modelle auf die Elemente zu reduzieren, die sich für ein bestimmtes Betrachtungskriterium, wie etwa ein Testziel, als wichtig erweisen.

Statt einer Anweisung und einer Menge von Variablen wie beim Programm-Slicing, besteht ein Kriterium für das Modell-Slicing aus einem Element e aller im Modell enthaltenen Elemente E [21]. Beim Slicing werden dann all jene Elemente extrahiert, die e auf irgendeine Art und Weise beeinflussen. Gleichzeitig muss dabei sichergestellt sein, dass der sich ergebende Slice ein funktionierendes und wohlgeformtes Modell bildet.



(a) State Machine M (b) Slice von M für $C = (s2)$

Abbildung 2.6.: State Machine eines Verkaufsautomaten vor und nach dem Slicing

Abbildung 2.6a zeigt eine einfache State Machine für einen Verkaufsautomaten, der nur Kaffee verkauft. Nach Auswahl des Getränks wird entweder Kaffee ausgegeben oder ein Ton abgespielt, falls ein Fehler auftritt - etwa weil der Kaffee leer ist. In Abbildung 2.6b ist der Slice von M abgebildet, der sich für das Kriterium $C = (s2)$ ergibt. Hierbei wurden die Zustände `s3` und `s4` sowie die Transitionen `t4` und `t5` entfernt, da diese keinerlei Auswirkungen auf `s2` haben.

Ein Slice beim Modell-Slicing ergibt sich dabei aus den Abhängigkeiten Dep_M, die zwischen den Elementen einer State Machine bestehen [21]. Diese Abhängigkeiten gliedern sich in:

- **Parallele Abhängigkeit:** Diese Abhängigkeit besteht zwischen simultan laufenden Elementen, wie sie bespielsweise bei parallelen Substate Machines vorkommen.
- **Sequentielle Datenabhängigkeit:** Von dieser Abhängigkeit sind Elemente betroffen, die sequentiell geordnet sind und auf die selben Variablen zugreifen.
- **Parallele Datenabhängigkeit:** Diese Abhängigkeit gilt für simultan laufende Elemente, welche auf die selben Variablen zugreifen.
- **Synchronisationsabhängigkeit:** Synchronisationsabhängigkeit besteht, wenn bei Elementen, die simultan laufen, das eine Element ein Event erzeugt, auf welches das andere Element reagiert.
- **Transitionskontrollabhängigkeit:** Diese Abhängigkeit gilt für sequentiell geordnete Elemente, bei denen ein Element ein Event erzeugt, welches von dem anderen Element benötigt wird.
- **Globale Kontrollabhängigkeit:** Globale Kontrollabhängigkeit findet sich zwischen Zuständen und Transitionen, wenn der Zustand der Ausgangszustand der Transition ist und die Transition durch ein Eingangsereignis ausgelöst wird.
- **Verfeinerungskontrollabhängigkeit:** Diese Abhängigkeit existiert zwischen einem Zustand und den Startzuständen all seiner Substate Machines.

Mithilfe der Abhängigkeiten wird ein Slice für eine State Machine erstellt, indem iterativ alle Elemente zum Slice hinzugefügt werden, bei denen eine Abhängigkeit zu einem Element im Slice besteht [21]. Der Slice besteht dabei zu Beginn nur aus dem Element e des Slicing-Kriteriums. Nach jeder Iteration wird außerdem überprüft, ob der entstandene Slice ein wohlgeformtes Modell bildet. Sollte dies nicht der Fall sein, werden weitere Elemente hinzugefügt, sodass die Wohlgeformtheit sichergestellt ist. So müssen beispielsweise für eine Transition sowohl Ausgangs- als auch Endzustand im Modell vorhanden sein. Das bedeutet, für eine durch Abhängigkeit hinzugefügte Transition, bei der einer der beiden Zustände noch im Slice fehlt, muss dieser in der selben Iteration hinzugefügt werden. Sobald die Iteration einen Fixpunkt erreicht hat, wird die entstandene State Machine noch einmal auf Wohlgeformheit geprüft. Hierbei werden dann etwa Zustände entfernt, zu denen zwar Abhängigkeiten bestehen, die aber durch keine Transitionen erreicht werden. Das Ergebnis dieser Überprüfung ist der resultierende Slice für das gewählte Kriterium.

Durch Verwendung von Modell-Slicing können Modelle, wie etwa State Machines, somit für ein bestimmtes Kriterium auf relevante Modellinhalte reduziert werden.

Slicing von Feature-annotierten State Machines

Diese bisher vorgestellte Form des Modell-Slicings bietet allerdings keine Möglichkeit den Anforderungen gerecht zu werden, die sich für Feature-annotierte State Machines ergeben [21]. So lässt sich zwar für ein bestimmtes Element als Kriterium der Slice über alle abhängigen Elemente erstellen, jedoch gibt es keine Option, um nur die Elemente in den Slice aufzunehmen, die etwa für ein

bestimmtes Feature gelten. Aus diesem Grund kombinieren Kamischke et al. [21] Modell-Slicing mit bedingtem Slicing um ein Feature-orientiertes Slicing zu ermöglichen.

Zur Anpassung von Modell-Slicing an Feature-orientierte State Machines werden der Algorithmus für einfaches Modell-Slicing sowie das einfache Slicing-Kriterium erweitert [21]. Um ein Slicing von Feature-annotierten State Machines (M, α) zu ermöglichen, wird hierbei das Modell-Slicing-Kriterium $C = (e)$ auf ein bedingtes Kriterium $C = (e, \phi)$ ausgeweitet. Als Bedingung ϕ wird beim Feature-orientierten Slicing entsprechend eine Feature-Konfiguration $\Gamma \in PC_{FM}$ in Form einer aussagenlogischen Formel über Feature-Parameter verwendet.

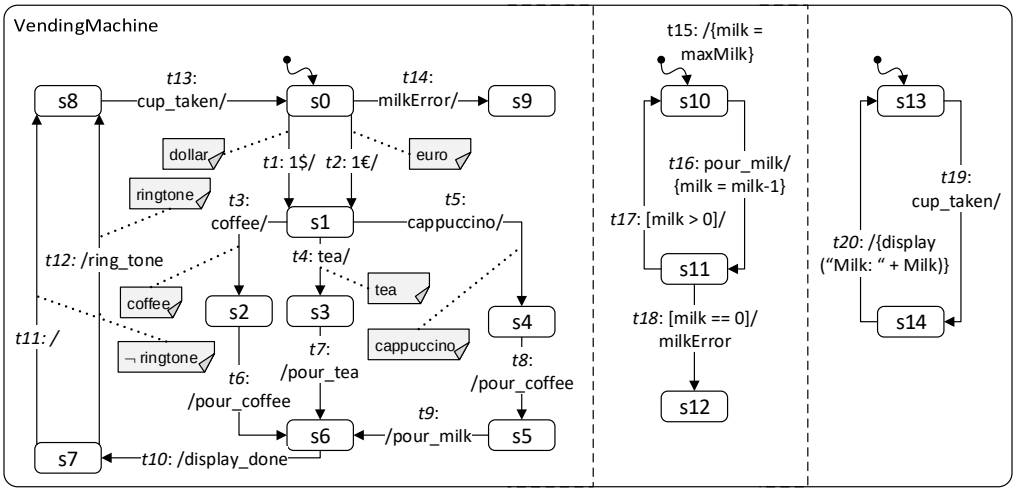


Abbildung 2.7.: Feature-annotierte State Machine (M, α) eines Verkaufsautomaten (angelehnt an [21])

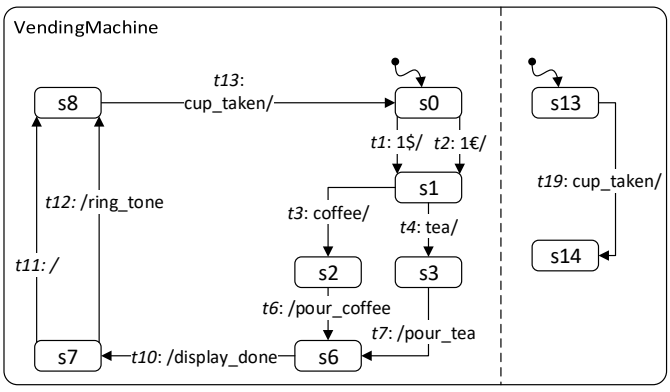


Abbildung 2.8.: Slice von (M, α) für $\Gamma = \{cappuccino \mapsto false\}$ (angelehnt an [21])

Abbildung 2.7 zeigt die Feature-annotierte State Machine eines erweiterten Verkaufsautomaten, welche in drei Substate Machines aufgeteilt ist. Als Beispiel, welches an Kamischke et al. [21] angelehnt ist, soll an dieser ein Slicing für eine Konfiguration $\Gamma = \{cappuccino \mapsto false\}$ durchgeführt werden, wobei vorerst $e = \varepsilon$ - das heißt also kein Zustand - für das Kriterium ausgewählt wird. In diesem Fall wird automatisch der Startzustand als erstes Element in den initialen Slice gewählt. Da

es sich bei Γ um eine Konfiguration ohne Cappuccino handelt, wird zunächst die mit `cappuccino` annotierte Transition `t5` entfernt. Als Folge kann Zustand `s4` entfernt werden, da dieser keine eingehende Transition mehr besitzt, und ebenso `t8`, `s5` und `t9`. Mit `t9` wurde die einzige Transition entfernt, welche die Aktion `pour_milk` auslöst. Daher wird auch `t16` entfernt, da dieser Übergang nun nicht mehr ausgelöst werden kann. Folglich werden `s11`, `t17`, `t18` und `s12` entfernt. Aufgrund des Verschwindens der Aktion `milkError` durch das Entfernen von `t18`, wird `t14` und somit auch `s9` gelöscht. Des Weiteren wird die Substate Machine, welche `s10` enthält, komplett entfernt, da sie nur noch den Startzustand und somit kein Verhalten mehr enthält. Außerdem wird `t20` gelöscht, da die Variable `Milk` nirgendwo mehr gesetzt wird. Als Slice für $\Gamma = \{cappuccino \mapsto false\}$ ergibt sich somit die State Machine in Abbildung 2.8. Wird für das Slicing-Kriterium nun zusätzlich Zustand `s13` als e gewählt, dann fallen außerdem Transition `t19` und Zustand `s14` weg, da diese `s13` nicht beeinflussen.

Kamischke et al. [21] erweitern für das Feature-orientierte Slicing das im letzten Abschnitt für Modell-Slicing vorgestellte Vorgehen. Daraus ergibt sich Algorithmus 2.5.

Algorithmus 2.5 .: 150%-State Machine Slicing Algorithmus

Input : 150%-State Machine $M_{150} = (M, \alpha)$, feature-orientiertes Slicing-Kriterium $C = (e, \Gamma)$
Output : Slice M_C

1 $Dep_M := computeDep(M_{150})$;
2 $M_0 := initSlice(M_{150}, C)$;
3 **repeat**
4 $\quad M_{i+1} = reachable(M'_i, Dep_M, \Gamma)$;
5 $\quad M'_{i+1} = wellformed^{(+)}(M_{i+1})$;
6 **until** $M'_{i+1} = M'_i$;
7 $M_C := wellformed^{(-)}(M'_{i+1})$;

Bei diesem Algorithmus werden eine Feature-annotierte State Machine (M, α) und ein Slicing-Kriterium $C = (e, \Gamma)$ mit $\Gamma \in PC_{FM}$ als Eingaben gewählt [21]. Weiterhin werden zunächst die Abhängigkeiten zwischen den Elementen durch $computeDep(M)$ bestimmt und der initiale Slice definiert, der vorerst nur e enthält. Wie zuvor beschrieben, werden durch Zeile 4 iterativ weitere Elemente hinzufügt, von denen die bisher im Slice enthaltenen Elemente abhängig sind. Der Unterschied zum normalen Modell-Slicing besteht nun darin, dass hierbei nur noch solche Elemente e hinzugefügt werden, für die gilt $\Gamma \vDash \alpha(e)$. Die Selektionsbedingung der hinzugefügten Elemente wird also durch die Konfiguration Γ erfüllt. Ferner werden wie zuvor weitere Elemente durch Zeile 5 hinzugefügt, damit nach jeder Iteration die Wohlgeformtheit des Modells garantiert ist. Das bedeutet, für jede Transition müssen Ausgangs- und Zielzustand existieren, jeder Zustand muss über mindestens einen Pfad vom Startzustand aus erreichbar sein und für einen Zustand s einer Substate Machine muss jeder Zustand s' existieren, in dem s enthalten ist. Nachdem der Fixpunkt des Slices - das heißt, wenn sich der Slice durch die Iteration nicht mehr verändert - in Zeile 6 erreicht ist, werden in Zeile 7, wie beim normalen Modell-Slicing auch, überflüssige Elemente entfernt um die Wohlgeformheit sicherzustellen. Dies ist ohne weitere Beachtung des Kriteriums durchführbar, da ein Erstellen von wohlgeformten State Machines immer möglich ist, indem lediglich weitere Elemente hinzugefügt werden, die ebenfalls das Kriterium erfüllen. Als Ausgabe des Algorithmus

ergibt sich dann der Slice M_C, der eine wohlgeformte State Machine für die Feature-Konfiguration Γ und das Element e des Slicing-Kriteriums C bildet.

Feature-orientiertes Slicing ist somit eine Möglichkeit der Komplexität in Produktmodellen Herr zu werden, die sich durch die Variabilität von Softwareproduktlinien ergibt.

2.4. Evolution von Softwareproduktlinien

Neben der bisher betrachteten *räumlichen Variabilität* von Softwareproduktlinien existiert in Form der *Evolution* zusätzlich eine *zeitliche Variabilität* [33]. Evolution bezeichnet die Veränderung von Produktlinien und deren Artefakten über die Zeit. Dadurch können zu unterschiedlichen Zeitpunkten verschiedene Versionen ein und desselben Artefakts vorliegen.

Evolution kann bewusst hervorgerufen werden, etwa wenn aufgrund von Innovationen oder Konkurrenzdruck der Softwareproduktlinie neue Funktionen hinzugefügt werden sollen [27]. Ebenso kann Evolution aber auch unerwünscht sein oder unerwartet auftreten. Dies kann beispielsweise der Fall sein, wenn Fehler gefunden werden, die dringend behoben werden müssen, oder bestimmte Funktionen nicht mehr unterstützt werden können. Während Evolution an sogenannten Variationspunkten, die schon unterschiedliche Wahlmöglichkeiten bieten, relativ einfach umgesetzt werden kann, ist es im Allgemeinen meist komplizierter, Änderungen ohne Schwierigkeiten umzusetzen. Dies rührt daher, dass verschiedene Bausteine häufig voneinander abhängig sind oder sich gegenseitig beeinflussen. Infolgedessen können Änderungen an einem Baustein immer auch Änderungen an einem abhängigen Baustein nach sich ziehen.

Evolution birgt aus diesem Grund immer auch Risiken, da beispielsweise bei einer widersprüchlichen Entwicklung von abhängigen Bausteinen Inkonsistenzen in der Softwareproduktlinie entstehen können [27]. Risiken ergeben sich vorallem bei der Störung der folgenden drei Kriterien für eine erfolgreiche Softwareproduktlinie:

- **Korrektheit:** Jede Veränderung an der Produktlinie kann zu einer Beeinträchtigung der Korrektheit führen, da immer die Möglichkeit besteht, dass die betroffenen Bausteine beschädigt werden, statt wie gewünscht optimiert zu werden.
- **Vollständigkeit:** Vorallem Änderungen an einer großen Anzahl von Bausteinen können eine unvollständige SPL zur Folge haben, da hierbei Verbindungen zwischen den Bausteinen verloren gehen können. Auf diese Weise kann es passieren, dass Bausteine keine Verbindung mehr zum Rest der Produktlinie aufweisen und somit entfernt werden. Eine andere Konsequenz ist, dass die Bausteine durch die fehlenden Verbindungen nicht mehr allen Konfigurationen zugeordnet werden können, zu denen sie eigentlich gehören, und in diesen dann nicht mehr auftauchen.
- **Konsistenz:** Werden voneinander abhängige oder sich gegenseitig beeinflussende Bausteine auf sich widersprechende Weise entwickelt, wird die Konsistenz der Produktlinie gestört. Das Resultat hierbei sind inkompatible Bausteine, durch die ein fehlerfreies Zusammenspiel der SPL unmöglich wird.

Jedes dieser Risiken kann bei Auftreten das Scheitern der kompletten Softwareproduktlinie zur Folge haben [27]. Dabei wird die Auftretenswahrscheinlichkeit mit wachsender Anzahl an Softwarebausteinen und steigender Komplexität der Beziehungen zwischen diesen stark erhöht.

Damit solche Risiken weitestgehend vermieden werden können, müssen die Effekte der Evolution auf die SPL bereits vor der eigentlichen Evolution bestimmt und im Evolutionsplan festgehalten werden [27]. Um die Auswirkungen, welche durch eine Änderung an der Produktlinie verursacht werden, zu identifizieren, kann besipielsweise die sogenannte *Change Impact Analyse* verwendet werden. Bei dieser Methode wird als erstes der Baustein betrachtet, der effektiv verändert werden soll. Als nächstes werden all jene Bausteine bestimmt, welche auf die eine oder andere Weise mit dem ersten Baustein in Beziehung bestehen. Diese werden dann daraufhin untersucht, ob die Änderung an dem ersten Baustein sie beeinflusst, sodass sie ebenfalls verändert werden müssen. Für jeden Baustein, der hierdurch ebenfalls als zu verändernder Baustein identifiziert wird, wird das Vorgehen wie für den ersten Baustein iterativ wiederholt. Durch dieses Vorgehen kann im Voraus bereits der komplette Effekt auf die gesamte Produktlinie festgestellt werden, den einzelne Veränderungen nach sich ziehen. So lässt sich erkennen, ob Konsistenz, Vollständigkeit und Korrektheit der SPL trotz der Evolution gewahrt bleiben und ob das gewünschte Ergebnis in einem positiven Verhältnis zum dafür betriebenen Aufwand steht.

Ferner ist bei der Betrachtung von Evolution nicht nur die Zukunft sondern auch die Vergangenheit der Produktlinie von Interesse [6]. Das Aufzeichnen der sogenannten *Evolutionshistorie* erlaubt die Untersuchung der erfolgten Veränderungen in der SPL über die Zeit. Diese Analyse lässt Rückschlüsse auf die mögliche, zukünftige Entwicklungsrichtung der Softwareproduktlinie zu. Des Weiteren gestattet sie die Wartung und Reparatur von älteren, sich noch im Umlauf befindlichen Versionen.

Überdies ist es, wie beim initialen Entwicklungsprozess, auch beim Evolutionsprozess hilfreich, die umzusetzenden Änderungen während jedes Schrittes dokumentiert vorliegen zu haben, um Fehler bei der Implementierung zu vermeiden. Damit also die Auswirkung der Evolution auf die SPL antizipiert, die Historie dokumentiert und der Evolutionsprozess fehlerlos erfolgen kann, müssen geeignete Methoden zur Verfügung stehen, Evolution in Softwareproduktlinien zu modellieren.

Wie auch die räumliche Variabilität der SPL kann Evolution auf zwei unterschiedlichen Abstraktionsebenen betrachtet werden: Zum einen auf Ebene der Produktlinie als Ganzem und zum anderen auf Ebene der Produkte selbst, etwa auf Artefaktlevel.

Evolution in Feature-Modellen

Auf der Produktlinienebene kann etwa das Konzept der Feature-Modelle erweitert werden, um auf diese Weise die zeitliche Variabilität darstellen zu können. Acher et al. [1] stellen beispielsweise einen Ansatz vor, der in Form eines Difference Models die semantischen und syntaktischen Unterschiede zwischen verschiedenen Feature-Modellen aufzeigt. In *Difference Models* werden die Unterschiede zwischen zwei Versionen eines Modells, ähnlich wie bei der Delta-Modellierung, durch hinzugefügte, entfernte und modifizierte Modellinhalte beschrieben. Alves et al. [2] nutzen Change Operators für Feature-Modelle, um auch ohne semantische Änderungen ein Refactoring von Softwareproduktlinien vornehmen zu können. *Change Operators* verhalten sich ähnlich wie Difference Models, jedoch können hierbei die Änderungen durch wesentlich komplexere Operatoren beschrieben werden. Beide Ansätze sind nicht explizit auf Evolution ausgelegt, sondern sollen zur generellen Beschreibung von Variabilität in SPLs dienen. Dennoch können diese Ansätze durchaus auch zur Darstellung von zeitlicher Variabilität dienen, wenn zeitlich aufeinanderfolgende Versionen von Feature-Modellen miteinander verglichen werden. Die Grundzüge von Difference Model- und

Change Operator-Ansätzen lassen sich allerdings nicht nur für Feature-Modelle, sondern für jegliche Art von Modell anwenden. Ansätze, die ausdrücklich auf die Darstellung von Evolution in Feature-Modellen ausgelegt sind, sind etwa EvoPL von Botterweck et al. [32], Temporale Feature-Modelle von Nieke et al. [31] und Hyper Feature-Modelle von Seidl et al. [36]. Diese drei Ansätze werden im Folgenden näher erläutert.

EvoFM

Das von Botterweck et al. propagierte Konzept des *EvoPL* [32] kombiniert die Prinzipien von Difference Models und Change Operators, um Evolution von Softwareproduktlinien auf Basis von Feature-Modellen zu modellieren [6]. Dabei soll der Ansatz die Möglichkeit bieten, sowohl die Evolution für die Zukunft zu planen als auch die vergangene Evolution analytisch zu betrachten.

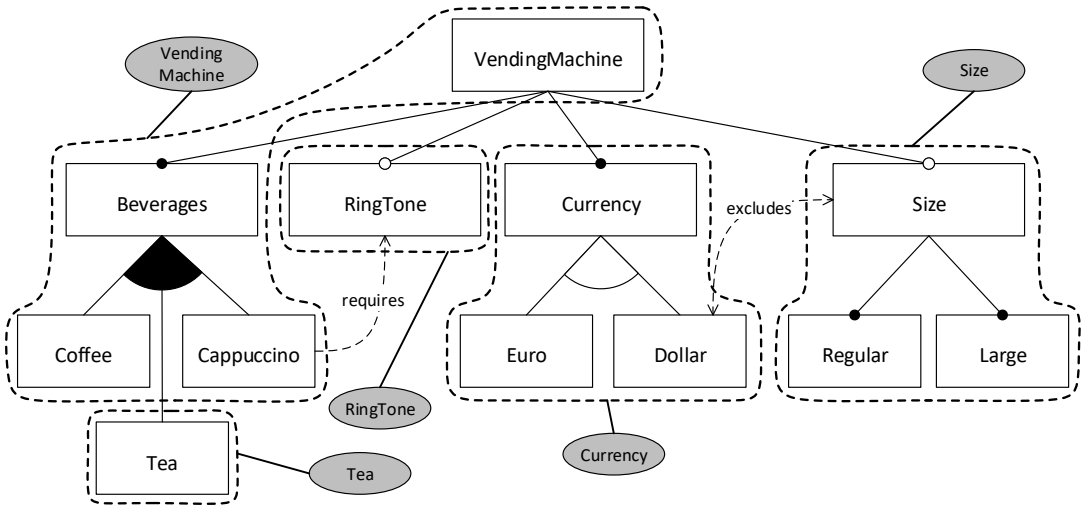


Abbildung 2.9.: Feature-Modell eingeteilt in Fragmente

Das EvoPL-Konzept basiert auf einer speziellen Art von Feature-Modell, dem *EvoFM* [7], welches aus Feature-Modell-Fragmenten und sogenannten evoOperatoren besteht [32]. Die Feature-Modell-Fragmente kapseln zusammenhängende Elemente des Feature-Modells, welche grundsätzlich nur gemeinsam während eines Evolutionschrittes hinzugefügt oder entfernt werden. Abbildung 2.9 zeigt eine solche Einteilung eines Feature-Modells in Fragmente. Jedem Fragment wird ein eindeutiger Name sowie ein Kontextelement zugeordnet. Letzteres dient zur Kennzeichnung der Position des Fragments im EvoFM. Ein Feature-Modell zu einem bestimmten Zeitpunkt in der Evolutionshistorie lässt sich somit als eine Zusammensetzung von bestimmten Fragmenten beschreiben. Die *evoOperatoren* sind den Fragmenten zugeordnete Change Operators, welche die Veränderungen innerhalb der Fragmente charakterisieren. Auf diese Weise kann etwa der Wechsel eines Features von optional auf mandatory oder umgekehrt sowie das Hinzufügen oder Entfernen eines Cross-Tree Constraints dargestellt werden. Das EvoFM spezifiziert dabei die Hierarchie und Abhängigkeiten der Fragmente und evoOperatoren.

Die einzelnen Evolutionsschritte lassen sich, ähnlich einer Feature-Konfiguration bei einem normalen Feature-Modell, durch eine Auswahl an Fragmenten und evoOperatoren repräsentieren [6]. Eine gültige EvoFM-Konfiguration bildet dementsprechend ein Feature-Modell für einen bestimm-

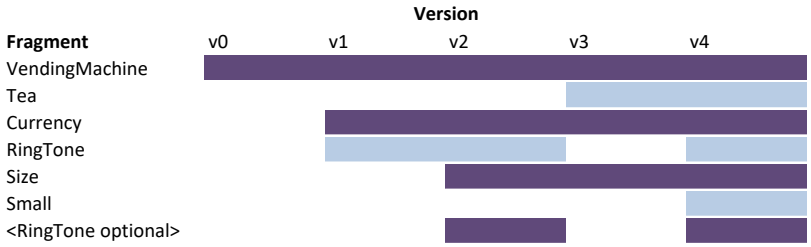


Abbildung 2.10.: Evolutionsplan

ten Zeitpunkt in der Evolutionshistorie. Die komplette Historie kann folglich durch eine Sequenz von EvoFM-Konfigurationen dargestellt werden. Durch einen sogenannten *Evolutionsplan* kann dabei definiert werden, welche EvoFM-Konfiguration in welchem Evolutionsschritt beziehungsweise welcher Version der Produktlinie gültig ist. Abbildung 2.10 zeigt einen Evolutionsplan für das in Fragmente eingeteilte Feature-Modell aus Abbildung 2.9. Der Plan zeigt, welche Fragmente in welcher Feature-Modell-Version gültig sind. Die Spalten beschreiben dementsprechend die genaue Zusammensetzung des Feature-Modells für die jeweilige Version. In der letzten Zeile findet sich der EvoOperator <`RingTone optional`>, der eine Veränderung der Variabilitätseigenschaft des Features `RingTone` ab Version `v2` anzeigt.

Temporale Feature-Modelle

Im Gegensatz zur Fragmentierung der Feature-Modelle wie beim EvoFm von Botterweck et al. [32], nutzen die Temporalen Feature-Modelle (TFM) von Nieke et al. [31] Zeitstempel für jeden einzelnen Modellinhalt. Diese Zeitstempel definieren die Gültigkeit der Modellinhalte, um so die Evolution von Feature-Modellen festhalten zu können. Auf diese Weise soll ebenfalls verhindert werden, dass bei der Veränderung des Feature-Modells während des Evolutionsprozesses die Informationen über ältere Modellversionen verloren gehen.

Der Kern der Temporalen Feature-Modelle sind sogenannte *Evolving Elements* [31]. Hierbei handelt es sich um Modellelemente, die sich über die Zeit entwickeln können und somit nur während einer limitierten Zeitspanne gültig sind. Jedem Element ist daher eine temporale Validität ϑ zugeordnet. Bei der temporalen Validität handelt es sich um ein rechtsoffenes Intervall $[\vartheta_{since}, \vartheta_{until})$. ϑ_{since} ist dabei der Zeitpunkt, an dem die temporale Validität beginnt, während ϑ_{until} das Ende der temporalen Validität markiert. Da es sich um ein rechtsoffenes Intervall handelt, ist das Element zum Zeitpunkt ϑ_{until} nicht mehr gültig, sondern nur bis zum Zeitpunkt $\vartheta_{until} - 1$. Soll ein Element zu einem bestimmten Zeitpunkt gelöscht werden, muss ϑ_{until} also auf genau diesen Zeitpunkt gesetzt werden.

In Temporalen Feature-Modellen werden neben den eigentlichen Features auch alle Eigenschaften und Zusammenhänge, die sich durch Evolution auf irgendeine Weise ändern können, als Evolving Elements definiert [31]. Jedes Evolving Element wird als eigene Entität im Temporalen Feature-Modell festgehalten. Die Evolving Elements zusätzlich zu den Features sind:

- **Gruppen:** Dabei handelt es sich um Alternativ-, Or- oder ungebundene Gruppen, die zusammen die untergeordneten Features für ein Eltern-Feature bilden. Aus ungebundenen Gruppen kann unter Berücksichtigung von Mandatory- und Optional-Features eine beliebige An-

zahl an Features ausgewählt werden. Besitzt ein Feature nur ein einziges untergeordnetes Feature, handelt es sich bei diesem einzelnen Feature trotzdem ebenfalls um eine Gruppe. Im Temporalen Feature-Modell wird die Gruppe als eigene Entität festgehalten, der die verschiedenen Features zugeordnet werden. Auf diese Weise wird das Erstellen und Löschen von Gruppen ermöglicht.

- **Feature-Namen:** Die Namen von Features werden als eigene Entitäten im TFM repräsentiert, um das Umbenennen von Features zu ermöglichen.
- **Kardinalitäten:** Kardinalitäten existieren sowohl für einzelne Features als auch für Gruppen und geben den Grad der Variabilität an.
  - **(1..1)** für Mandatory-Features
  - **(0..1)** für Optional-Features
  - **(1..1)** für Alternative-Gruppen
  - **(1..*n*)** für Or-Gruppen
  - **(0..*n*)** für ungebundene Gruppen

  Die Existenz von Kardinalitäten als eigene Entitäten erlaubt die Veränderung der Variabilitätseigenschaften der Features und Gruppen.
- **Feature-Attribute:** Mögliche Attribute von Features werden ebenfalls als eigene Entitäten festgehalten, um das Erstellen, Umbenennen und Löschen von Attributen zu ermöglichen.
- **Cross-Tree Constraints:** Cross-Tree Constraints werden in Form von aussagenlogischen Formeln als eigene Entitäten repräsentiert. Auf diese Weise besteht die Möglichkeit diese zu erstellen oder zu löschen. Die Veränderung von Teilausdrücken der Constraints ist nicht erlaubt, stattdessen müssen Constraints in diesem Fall komplett gelöscht und wieder neu erstellt werden.
- **Kind-Features:** Die Eltern-Kind-Relation zwischen einem Feature und seinen untergeordneten Features wird durch Kind-Features dargestellt. Jedem Kind-Feature wird genau eine Gruppe zugeordnet, wodurch die Position der Gruppe im Feature-Modell eindeutig bestimmt wird. Ein Feature kann mehrere Kind-Features besitzen. Das Vorhandensein von Kind-Features als eigene Entitäten im Modell erlaubt das Verschieben von Gruppen.
- **Gruppen-Kompositionen:** Da Gruppen sich im Verlauf der Evolution verändern können, existieren für jede Gruppe genau so viele Gruppen-Kompositionen, wie es unterschiedliche Ausprägungen der Gruppe gibt. Einer Gruppen-Komposition ist dabei immer wenigstens ein Feature zugeordnet. Durch die Zuordnung von Features zu einer Gruppen-Komposition und somit zu einer bestimmten Gruppe, wird die Position der Features im Feature-Modell definiert. Die Darstellung von Gruppen-Kompositionen als eigene Entitäten ermöglicht das Verschieben von Features und die Änderung der Zusammenstellung von Gruppen.

Die Evolutionshistorie ist bei Temporalen Feature-Modellen durch die Temporalen Validitäten festgehalten [31]. Anhand der Validitäten lässt sich ablesen, welche Version des Feature-Modells zu welchem Zeitpunkt gültig ist, ohne dass die genaue Art der Änderungen zwischen zwei Versionen

bekannt sein muss. Zu Analysezwecken ist es allerdings von Nutzen, ebenjene Informationen über die vorgenommenen Modifikationen zu besitzen. Hierzu können die gesuchten *Evolutionsoperationen* anhand der Veränderungen von Temporalen Validitäten abgeleitet werden. Um die Evolutionsoperation für ein Element zwischen zwei Zeitpunkten τ_1 und τ_2 herzuleiten, werden ϑ_{since} und ϑ_{until} des Elementes zu beiden Zeitpunkten betrachtet. Es existieren fünf atomare Evolutionsoperationen, die sich auf die Evolving Elements anwenden lassen: *hinzufügen*, *löschen*, *umbenennen*, *bewegen* und *Typ ändern*. Der Typ bezieht sich hierbei auf die Variabilitätseigenschaft des Features. Davon ausgehend, dass gilt $\tau_1 \leq \tau_2$, lassen sich die Evolutionsoperationen durch die Temporalen Validitäten folgendermaßen definieren [31]:

- **Hinzufügen:**

  $$\tau_1 < \vartheta_{since} \leq \tau_2 < \vartheta_{until}$$

  Ein Element wurde somit hinzugefügt, wenn es zum Zeitpunkt τ_1 noch nicht existiert hat, aber zum Zeitpunkt τ_2 gültig ist.

- **Löschen:**

  $$\vartheta_{since} \leq \tau_1 < \vartheta_{until} \leq \tau_2$$

  Umgekehrt wurde ein Element gelöscht, wenn es zum Zeitpunkt τ_1 bereits vorhanden war, zum Zeitpunkt τ_2 aber nicht mehr gültig ist.

- **Umbenennen:**

  $$name_1 \neq name_2 \wedge \vartheta_{name_1 since} \leq \tau_1 < \vartheta_{name_1 until} \leq \vartheta_{name_2 since} \leq \tau_2 < \vartheta_{name_2 until}$$

  Ein Element wurde umbenannt, wenn zwei Namen $name_1$ und $name_2$ für das Element existieren. Dabei ist $name_1$ zum Zeitpunkt τ_1 gültig, jedoch nicht mehr zum Zeitpunkt τ_2. Stattdessen ist ab τ_2 $name_2$ gültig.

- **Typ ändern:**

  $$card_1 \neq card_2 \wedge \vartheta_{card_1 since} \leq \tau_1 < \vartheta_{card_1 until} \leq \vartheta_{card_2 since} \leq \tau_2 < \vartheta_{card_2 until}$$

  Verhält sich bei einer Typänderung mit den Kardinalitäten $card_1$ und $card_2$ analog zum Umbenennen.

- **Bewegen:** Beim Bewegen wird eine Fallunterscheidung vorgenommen:
  - **Feature:**

    $$group_1 \neq group_2 \wedge \vartheta_{compo_1 since} \leq \tau_1 < \vartheta_{compo_1 until} \leq \vartheta_{compo_2 since} \leq \tau_2 < \vartheta_{compo_2 until}$$

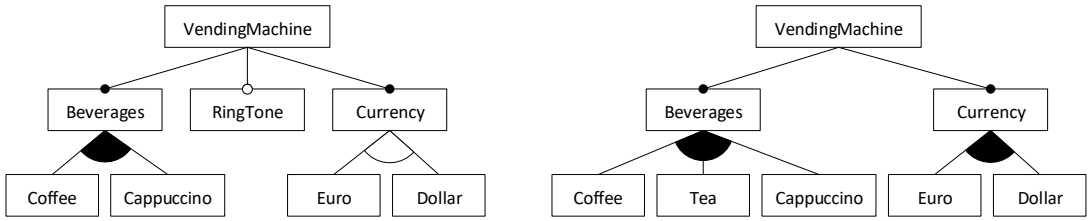
    Wurde ein Feature in eine Gruppe hinein oder aus einer Gruppe hinaus bewegt, dann ist das Element Teil zweier Gruppen-Kompositionen $compo_1$ und $compo_2$, die zwei unterschiedlichen Gruppen $group_1$ und $group_2$ zugeordnet sind. Da die Gruppen auch über die Verschiebung des Elements hinaus gültig bleiben können, werden die zeitlichen Validitäten der Gruppen-Kompositionen betrachtet. Hierbei ist $compo_1$ zum Zeitpunkt τ_1 gültig, allerdings nicht mehr zum Zeitpunkt τ_2, während ab τ_2 $compo_2$ gültig ist.

- **Gruppe:**

$$parentFeature_1 \neq parentFeature_2$$
$$\wedge$$
$$\vartheta_{featureChild_1 since} \leq \tau_1 < \vartheta_{featureChild_1 until} \leq \vartheta_{featureChild_2 since} \leq \tau_2 < \vartheta_{featureChild_2 until}$$

  Das Bewegen einer Gruppe verhält sich analog zum Bewegen eines Features, jedoch werden hierbei statt zweier unterschiedlicher Gruppen zwei unterschiedliche Eltern-Feature und statt der Validitäten der Gruppen-Kompositionen die Validitäten der Eltern-Kind-Beziehungen $featureChild_1$ und $featureChild_2$ betrachtet. Handelt es sich um ein einzelnes, eigenständiges Feature, das bewegt werden soll, wird dieses als Gruppe behandelt.

Um das Konzept der Temporalen Feature-Modelle zu veranschaulichen, zeigt Abbildung 2.11a das Feature-Modell eines Verkaufsautomaten zum Zeitpunkt τ_1 und Abbildung 2.11b zum Zeitpunkt τ_2. Zum Zeitpunkt τ_1 können die Getränke Kaffee und Cappuccino beliebig kombiniert werden. Des Weiteren muss eine von zwei Währungen gewählt werden und es besteht die Möglichkeit bei Getränkeausgabe einen Ton abspielen zu lassen. Zum Zeitpunkt τ_2 wurde das Getränk Tee hinzugefügt und das Feature `RingTone` entfernt. Darüber hinaus wurde aus der Alternative-Gruppe `Euro` und `Dollar` eine Or-Gruppe, sodass nun auch beide Währungen angeboten werden können.



(a) Feature-Modell zu Zeitpunkt τ_0 (b) Feature-Modell zu Zeitpunkt τ_2

Abbildung 2.11.: Feature-Modelle eines Verkaufsautomaten

Abbildung 2.12 visualisiert die Struktur eines Temporalen Feature-Modells, welches die beiden Feature-Modelle der Zeitpunkte τ_1 und τ_2 vereint. Die Features `Beverages` und `Currency` besitzen je die Kardinalität (1..1) für Mandatory-Features, während `RingTone` durch die Kardinalität (0..1) ein Optional-Feature ist. `Tea`, `Coffee` und `Cappuccino` gehören zu der Or-Gruppe `group_a` mit der Kardinalität (1..n). Die drei Features bilden dabei zwei unterschiedliche Kompositionen `compo_a1` und `compo_a2`. `compo_a1` besitzt die Temporale Validität $\vartheta = (\tau_1, \tau_2)$ und `compo_a2` die Validität $\vartheta = (\tau_2, > \tau_2)$. Das Feature `Tea` besitzt dabei ebenfalls die Validität $\vartheta = (\tau_2, > \tau_2)$. Alle Inhalte, für die, wie für `Cappuccino` oder `Coffee`, keine gesonderte Validität angegeben ist, besitzen die Default-Validität $\vartheta = (\tau_1, > \tau_2)$. Die Validität von `Tea` zeigt an, dass das Feature erst ab τ_2 existiert und somit zum Zeitpunkt τ_2 hinzugefügt worden sein muss. Ebenso lässt sich erkennen, dass sich die Gruppe `group_a` verändert hat, da für sie zwei unterschiedliche Kompositionen existieren: einmal `compo_a1` zum Zeitpunkt τ_1 und einmal `compo_a2` ab Zeitpunkt τ_2. Das Entfernen von Feature `RingTone` lässt sich daran erkennen, dass die Validität ab τ_2 nicht mehr gegeben ist. Die Gruppe `group_d`,

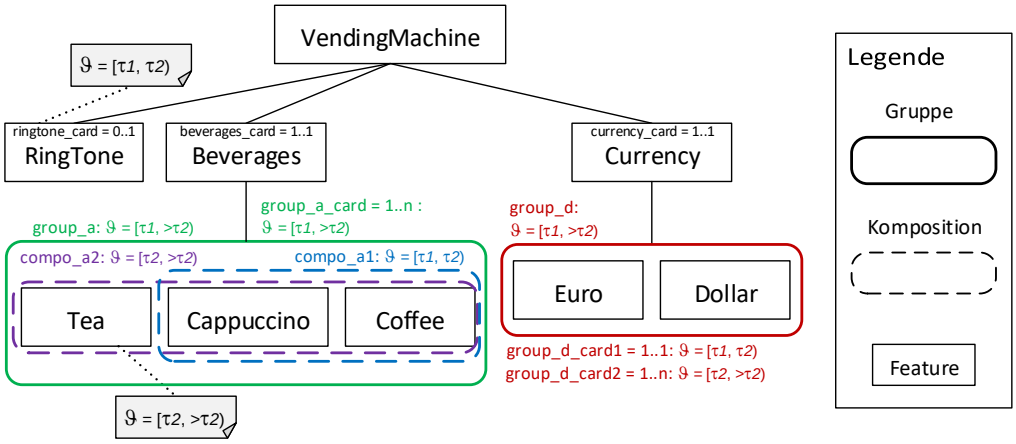


Abbildung 2.12.: Temporales Feature-Modell eines Verkaufsautomaten

welche die Features `Euro` und `Dollar` enthält, besitzt zwei Kardinalitäten `group_d_card1` = (1..1) und `group_d_card2` = (1..n) mit den unterschiedlichen Validitäten $\vartheta = (\tau_1, \tau_2)$ und $\vartheta = (\tau_2, > \tau_2)$. Daraus lässt sich ableiten, dass der Typ von `group_d` von Alternative-Gruppe auf Or-Gruppe geändert wurde. Die Komposition für `group_d` wird nicht gesondert dargestellt, da die Komposition sich während der Existenz der Gruppe nicht ändert.

Anhand der vorgenommenen oder vorzunehmenden Evolutionsoperationen bieten Temporale Feature-Modelle eine Change Impact Analyse mit der Möglichkeit wichtige Konfigurationen festzusetzen (sogenanntes *Configuration Locking*) [31]. Auf diese Weise soll bestimmt werden, ob die Änderungen dazu führen, dass bestimmte Konfigurationen nach der Evolution nicht mehr möglich sind. Dies erwiese sich als Problem, sollte es sich dabei um sehr wichtige Konfigurationen handeln, die von einer Vielzahl von Kunden benötigt werden. Bei der Change Impact Analyse werden die Auswirkungen der Veränderungen auf das Feature-Modell in sogenannten *Broken Configuration Categories* eingeordnet [31]:

- **Not Broken:** Die Konfiguration ist weiterhin gültig. Diese Kategorie wird in weitere Unterkategorien aufgeteilt, um mögliche Effekte der Evolution weiter zu spezifizieren:
  - **Refactoring:** Bei der Evolution wurde die SPL lediglich überarbeitet, ohne Änderungen an der Struktur vorzunehmen, wie beispielsweise bei der Umbennenung eines Features.
  - **Unaffected:** Die Konfiguration ist nicht von den Änderungen innerhalb der SPL betroffen, beispielsweise wenn ein Feature entfernt wurde, das nicht in der Kofiguration enthalten ist, oder wenn ein enthaltenes Feature einer Typänderung von optional zu mandatory unterzogen wurde.
  - **Extended:** Dies ist etwa der Fall, wenn neue Features zum Feature-Modell hinzugefügt wurden. Auch dadurch ergibt sich keine Änderung an der Konfiguration, aber es besteht die Möglichkeit die Konfiguration zu erweitern.
- **Outdated:** Bei der Konfiguration handelt es sich um eine veraltete, ungültige Version, da einige der enthaltenen Features nicht mehr existieren, z.B. weil sie gelöscht wurden.

- **Conflicting:** In diesem Fall widerspricht die Konfiguration dem aktuellen Feature-Modell, beispielweise durch die Typänderung einer Or-Gruppe in eine Alternative-Gruppe, wenn mehr als ein Feature der Gruppe in der Konfiguration enthalten ist.

Jede dieser Kategorien lässt sich explizit aus der genauen Beschaffenheit der Evolutionsoperation ableiten [31]. So ist etwa jeder speziellen Typänderung, beispielsweise von Mandatory-Feature zu Optional-Feature oder von ungebundener Gruppe zu Or-Gruppe, eine Kategorie zugeordnet. Um nun das Entwerten von festgesetzten Konfigurationen zu verhindern, können die vorzunehmenden Änderungen im Hinblick auf diese Konfigurationen der Change Impact Analyse unterzogen werden. Sollte sich für eine der festgesetzten Konfigurationen eine andere Kategorie als „Not Broken" ergeben, kann die Evolution auf diese Weise nicht stattfinden.

Hyper Feature-Modelle

Eine etwas andere Art der Darstellung von Evolution als die beiden bisher beschriebenen Ansätze, verfolgen die Hyper Feature-Modelle (HFMs) von Seidl et al. [36]. Während auch hier weiterhin Feature-Modelle die Grundlage bilden, wird nicht die Evolution des gesamten Modells sondern der einzelnen Features betrachtet.

Hyper Feature-Modelle dienen dazu, die Versionierung von Features innerhalb eines Feature-Modells zu unterstützen [36]. Das bedeutet, statt einer Versionierung des kompletten Feature-Modells, bei der Veränderungen an der Struktur des Modells vorgenommen werden, werden bei diesem Ansatz lediglich Versionen für die einzelnen Features unterstützt, während die Struktur des Modells beibehalten wird. Eine Version eines Features bezeichnet dabei eine Momentaufnahme von der inneren Umsetzung der Funktion des Features. Durch Evolution kann sich diese Umsetzung ändern, wodurch einzelne Features in unterschiedlichen Versionen vorhanden sein können. Eine neue Version ergibt sich durch die Änderungen allerdings nur, wenn die Umsetzung auf bedeutsame Weise geändert wurde. Das heißt, dass das Feature nicht nur einer einfachen Umstrukturierung ausgesetzt war, sondern dass etwa wesentliche Fehler ausgebessert wurden oder das Feature in seiner Funktion erweitert wurde. Jedoch muss sich auch bei einer bedeutsamen Änderung nicht zwangsläufig eine neue Version ergeben, wenn sich dadurch beispielsweise keine Auswirkungen auf das Zusammenspiel mit anderen Feature-Versionen ergeben.

Hyper Feature-Modelle nehmen sich damit den Problemen an, die sich vor allem in sogenannten Software Ökosystemen (SECOs) ergeben [36]. Bei diesen gibt es im Gegensatz zu Softwareproduktlinien kein zentrales Management mit einem synchronisierten Entwicklungszyklus, sondern die Bausteine können sich unabhängig voneinander entwickeln, wenn sie etwa von unterschiedlichen Zulieferern stammen. Auf diese Weise kann die aktuelle Version eines Features möglicherweise mit der vorherigen Version eines anderen Features kompatibel sein, mit der neuesten Version allerdings nicht. Durch HFMs sollen daher auch ältere Versionen von Features unterstützt werden, damit auch Kunden ihre Produkte noch nutzen können, die nicht auf neuere Versionen wechseln wollen oder können. Dies könnte der Fall sein, wenn ihr Produkt ein Feature enthält, das nicht mehr mit der neuesten Version eines anderen enthaltenen Features kompatibel ist.

Abbildung 2.13 veranschaulicht die grafische Darstellung eines Hyper Feature-Modells. Dieses ist dem Beispiel von Seidl et al. [36] nachempfunden. Das Hyper Feature-Modell zeigt die Versionierung der Features für den mobilen Roboter *TurtleBot*. Die Abbildung zeigt, dass eine Versionierung von Features nicht nur linear erfolgen kann, sondern dass auch eine Verzweigung von Features

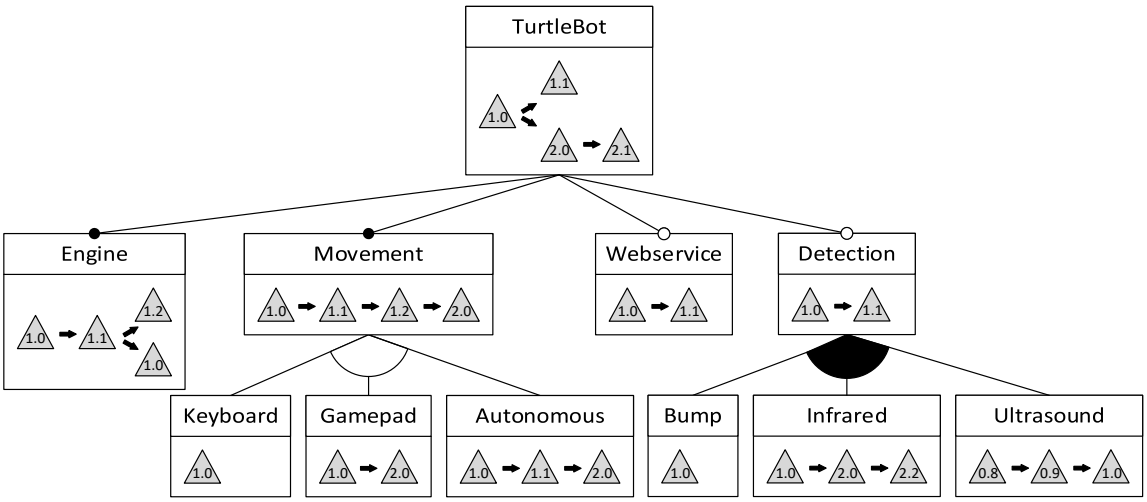


Abbildung 2.13.: Hyper Feature-Modell des mobilen Roboters *TurtleBot* (nach Seidl et al. [36])

möglich ist. Dies ist etwa der Fall, wenn die neueste Version auf einem anderen Vorgänger basiert als auf der zuvor aktuellsten Version. Dies ist etwa beim Feature `TurtleBot` der Fall, bei dem für Version 1.0 zuerst Version 1.1 nachfolgte und danach jedoch Version 2.0 entwickelt wurde, die wiederum auf Version 1.0 basiert.

Zusätzlich zu den Versionsinformationen der Features bieten Hyper Feature-Modelle versionssensible Cross-Tree Constraints [36]. Diese enthalten im Vergleich zu normalen Constraints Informationen darüber, welche Versionen von Features inkompatibel mit bestimmten Versionen von anderen Features sind oder welche Versionen für eine bestimmte Version benötigt werden. Mithilfe der versionssensiblen Constraints kann angegeben werden, dass eine Version nur mit den Versionen eines bestimmten Intervalls $[1.1 - 1.2]$ kompatibel ist oder dass sie mindestens Version $2.x$ oder neuer benötigt. Zusätzlich sind bedingte Cross-Tree Constraints möglich. Diese geben etwa an, dass für Version 1.2 oder größer von Feature `Movement` mindestens Version 1.1 von Feature `Detection` gewählt werden muss, *sofern* `Detection` für die Konfiguration gewählt wurde. Durch diese Constraints wird so verhindert, dass durch die normalen Cross-Tree Constraints suggeriert wird, dass für `Movement` mit einer Version größer 1.2 das Feature `Detection` gewählt werden *muss*.

Neben der Darstellung von Evolution auf Ebene der gesamten Produktlinie durch erweiterte Feature-Modelle, wie beispielsweise EvoFMs, Temporale Feature-Modelle oder Hyper Feature-Modelle, kann Evolution auch auf Artefaktlevel dargestellt werden, so etwa in Produktmodellen. Die Modellierung von Evolution in Produktmodellen wird im Folgenden näher erläutert.

Evolution in Produktmodellen

Evolution beschreibt eine Veränderung eines Modells über die Zeit und resultiert somit in einer natürlichen Transformation des Modells [27]. Dementsprechend häufig wird die Modellierung von Evolution auf Artefaktlevel durch transformationale Ansätze vorgenommen. So lassen sich etwa die Change Sets von Hendrickson et al. [18] - welche die Architekturen verschiedener Produkte spezifizieren, indem die Change Sets kombiniert werden - zusätzlich zur generellen Beschreibung von Variabilität auch für Evolution anwenden [6].

Des Weiteren basieren viele Ansätze auf Delta-Modellierung [10] (s. Kapitel 2.1). So erstellen etwa Lima et al. [23] Delta-Modelle, um Konflikte zu identifizieren, die sich durch die gegensätzliche Entwicklung von Code-Bausteinen ergeben, denen Features zugeordnet sind. Der Nachteil bei diesem Ansatz ist jedoch, dass immer nur ein Evolutionsschritt dargestellt werden kann. Dhungana et al. [15] verwenden ebenso Deltas zur Erfassung von Unterschieden zwischen verschiedenen Versionen von Softwareproduktlinien. Mithilfe der Deltas werden auch hier Konflikte aufgespürt und aufgelöst. Sowohl der Ansatz von Lima et al. als auch der Ansatz von Dhungana et al. ermöglichen eine Change Impact Analyse. Allerdings wird in beiden Fällen keine Methode geboten, welche räumliche und zeitliche Variabilität auf die gleiche Weise handhabt.

Im Gegensatz dazu bieten Seidl et al. [37] einen Ansatz, der sowohl die Darstellung der räumlichen Variabilität als auch der Evolution auf die selbe Weise durch Deltas umsetzt. Es handelt sich hierbei um eine generische Methode, die für unterschiedliche Artefakttypen angewendet werden kann. Der Nachteil ist allerdings, dass hierbei keine Change Impact Analyse möglich ist. Während Seidl et al. die Deltas für zeitliche und räumliche Variabilität beide auf der selben Modellierungsebene anwenden, werden bei der Higher-Order Delta-Modellierung von Lity et al. [25] die Deltas, welche die Evolution vornehmen, auf einer höheren Ebene angewendet. Da hierdurch eine Change Impact Analyse bei gleichzeitiger Nutzung des selben Vorgehens für Variabilität und Evolution möglich ist, wird Higher-Order Delta-Modellierung in dieser Arbeit zum Vergleich mit dem erarbeiteten Konzept zur Modellierung von Evolution in Feature-annotierten State Machines herangezogen.

Schließlich bieten Haber et al. [17] einen Ansatz, dessen Fokus auf SPL-Architekturen liegt und der durch Änderungen an Delta-Modellen die Modellierung von Variabilität in Zeit und Raum erlaubt. Während dieser Ansatz zur Verhinderung einer Degenerierung der Softwareproduktlinie bei der Evolution eine Refactoring-Möglichkeit zur Verfügung stellt, besteht jedoch keine Option zur Erfassung einer Evolutionshistorie.

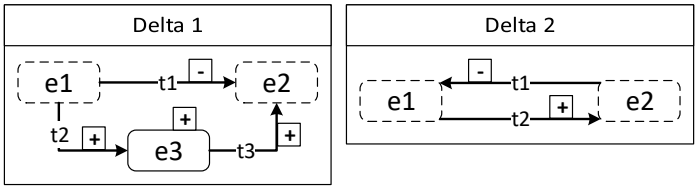
Während für transformationale Methoden also eine breite Masse an Möglichkeiten zur Modellierung von Evolution besteht, ergeben sich für annotative oder kompositionale Vorgehensweisen weitaus weniger Ansätze. So existieren etwa die Safe Evolution Templates von Neves et al. [30], welche sowohl für kompositionale als auch annotative Softwareproduktlinien angewendet werden können. Durch die Templates wird sichergestellt, dass keine existierenden Varianten durch die Anwendung von Evolutionsoperationen unerwünscht beeinflusst werden. Allerdings erfolgt auch hier die Behandlung von zeitlicher und räumlicher Variabilität nicht durch eine gemeinsame Methode. Eine Evolutionshistorie existiert bei den Safe Evolution Templates nur indirekt. Diese muss anhand der Menge von angewendeten Templates hergeleitet werden.

Eine Methode, die für annotative Ansätze, wie etwa Feature-annotierte State Machines [21], eine Möglichkeit bietet sowohl Variabilität als auch Evolution auf die gleiche Weise zu spezifieren, existiert demnach nicht. Aus diesem Grund wird in dieser Arbeit ein solcher Ansatz eingeführt. Da als Vergleich zur Evaluation des entwickelten Ansatzes Higher-Order Delta-Modellierung verwendet wird, wird diese im folgenden Abschnitt näher vorgestellt.

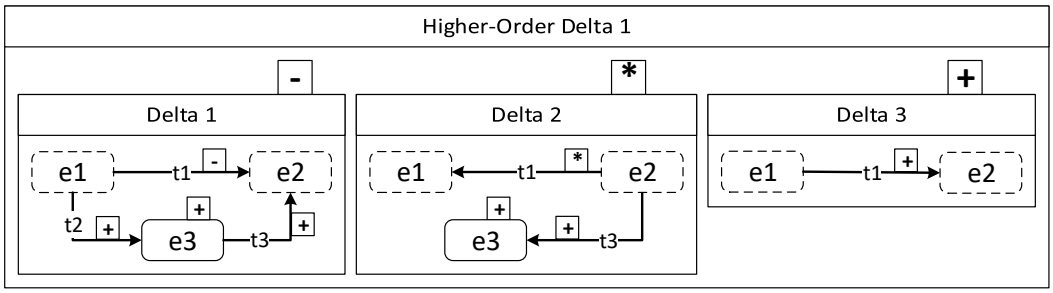
Higher-Order Delta-Modellierung

Higher-Order Delta-Modellierung basiert auf der Delta-Modellierung (s. Kapitel 2.1) und dient dazu, Evolution in Softwareproduktlinien zu modellieren [25]. Dabei ist Higher-Order Delta-Modellierung, ebenso wie die Delta-Modellierung, unabhängig von Programmier- oder Modellierspra-

chen. Während bei der Delta-Modellierung Deltas auf Kernmodelle angewendet werden, wird bei der Higher-Order Delta-Modellierung das Konzept auf die nächsthöhere Stufe gehoben, indem Higher-Order Deltas (HODs) auf komplette Delta-Modelle angewendet werden. Statt des Hinzufügens, Entfernens und Modifizierens von Modellelementen durch Deltas, werden hierbei durch die HODs komplette Deltas hinzugefügt, entfernt oder modifiziert. Ein Higher-Order Delta-Modell $DM^H = (DM, \Delta^H)$ besteht dementsprechend aus einem Delta-Modell DM und aus einer Menge $\Delta^H = \{\delta_0^H, ..., \delta_n^H\}$ von Higher-Order Deltas $\delta^H = \{op_1^H, ..., op_n^H\}$ mit Operationen $op^H = (\text{add } \delta / \text{rem } \delta / \text{ mod } (\delta, \{\text{add } op, ..., \text{rem } op', ...\}))$. Folglich wird durch ein Higher-Order Delta-Modell die Menge an Deltas in einem Delta-Modell verändert, was gleichzeitig eine Veränderung der aus dem Delta-Modell resultierenden Produktmodelle zur Folge hat.



(a) Delta-Menge



(b) Higher-Order Delta

Abbildung 2.14.: Beispiel für Higher-Order Delta-Modellierung

Die Abbildungen 2.14a und 2.14b verdeutlichen das Konzept der Higher-Order Delta-Modellierung. Abbildung 2.14a zeigt eine Delta-Menge bestehend aus den Deltas `Delta 1` und `Delta 2`. Indes bildet 2.14b das HOD `Higher-Order Delta 1` ab, welches Veränderungen an der Delta-Menge aus Abbildung 2.14a vornimmt. So wird `Delta 1` etwa komplett entfernt, während `Delta 3` neu zur Delta-Menge hinzugefügt wird und `Delta 2` modifiziert wird. Die Modifizierung eines Deltas äußert sich in der Veränderung der beinhalteten Operationen des Deltas [25]. So können etwa bestehende Operationen entfernt oder modifiziert als auch neue Operationen hinzugefügt werden. In Abbildung 2.14b wurde bei der Modifikation von `Delta 2` das Hinzufügen von **t2** entfernt als auch das Entfernen von `t1` modifiziert, sodass es sich nun um eine Modifikation handelt. Des Weiteren wurde das Hinzufügen von `t3` und `e3` ergänzt.

Higher-Order Delta-Modelle bieten die Möglichkeit die Evolutionshistorie einer Softwareproduktlinie festzuhalten [25]. Diese setzt sich aus vielen Evolutionsschritten zusammen, wobei ein Evolutionsschritt durch die Entwicklung eines Delta-Modells in die zeitlich darauffolgende Versi-

on beschrieben wird. Es existieren hierbei zwei unterschiedliche Arten von Higher-Order Delta-Modellen:

- **Evolution-Step Higher-Order Delta-Modell:** Evolution-Step HOD-Modelle bestehen aus dem zugrunde liegenden Delta-Modell und genau einem Higher-Order Delta. Dieses HOD beschreibt die vorgenommenen Veränderungen während des Evolutionsschrittes.
- **Evolution-History Higher-Order Delta-Modell:** Evolution-History HOD-Modelle setzen sich zusammen aus einem Delta-Modell und einer zeitlich geordneten Menge von Higher-Order Deltas. Die sich so ergebende Sequenz von HODs wird nacheinander auf das Delta-Modell angewendet. Derart ergibt sich die komplette Evolutionshistorie der SPL, mit der Möglichkeit, jeden Evolutionsschritt einzeln zu betrachten. Hierfür werden die einzelnen Higher-Order Deltas inkrementell auf das durch den Vorgänger erzeugte Delta-Modell angewendet, solange bis das Delta-Modell für den Schritt erreicht ist, der betrachtet werden soll.

Higher-Order Delta-Modellierung erlaubt außerdem Schlussfolgerungen (sogenanntes *Reasoning*) über die Auswirkungen der Evolution. Bei der Delta-Modellierung ergibt sich für das Umwandeln von einer Modellversion in eine andere eine spezifische Delta-Sequenz, das heißt eine eindeutige Reihenfolge von anzuwendenden Deltas. Durch die Anwendung von HODs wird die Delta-Menge des Delta-Modells verändert und dementsprechend auch die Delta-Sequenzen, welche die betroffenen Deltas beinhalten. Eine Änderung einer Delta-Sequenz hat wiederum Auswirkungen auf das aus der Anwendung der Sequenz folgende Modell. Durch die Analyse der Änderungen an der Delta-Menge, also des Hinzufügens, Entfernens und Modifizierens von Deltas, können die Effekte auf die bestehenden Delta-Sequenzen abgeleitet werden. Anhand dieser Informationen lassen sich wiederum die Folgen für die Menge der erzeugbaren Produktmodelle ableiten. Higher-Order Delta-Modellierung ermöglicht somit, ein Delta-Modell zu verändern und gleichzeitig durch Analyse dieser Änderungen die Auswirkungen auf die Menge der resultierenden Produktmodelle zu kennen.

Im weiteren Verlauf dieser Arbeit wird Higher-Order Delta-Modellierung zum Vergleich mit der in Kapitel 3 entwickelten Methode zur Darstellung von Evolution in Feature-annotierten State Machines verwendet.

Nachdem nun die zum Verständnis dieser Arbeit benötigten Grundlagen erläutert wurden, wird im folgenden Kapitel ein Konzept vorgestellt, welches Feature-annotierte State Machines (s. Kapitel 2.2) erweitert, sodass diese auch Informationen über die Evolution einer Softwareproduktlinie darstellen können.

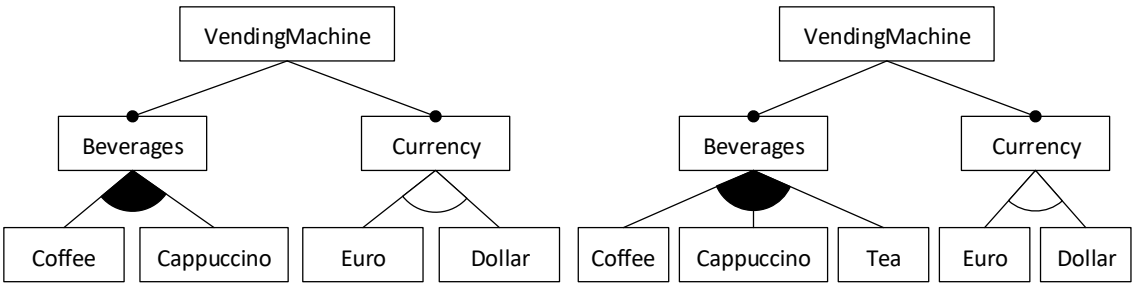
3 Evolution von Feature-annotierten State Machines

In diesem Kapitel wird ein Konzept zur Modellierung von Evolution in Feature-annotierten State Machines vorgestellt. Dazu wird zunächst eine geeignete Methodik zur Modellierung erarbeitet und diese anschließend formal definiert. Des Weiteren wird ein Algorithmus zum Slicing der erweiterten, annotierten State Machines eingeführt, um die Möglichkeit zu bieten, Analysen an diesen durchzuführen.

3.1. Konzeption der Darstellung von 175%-Modellen

Wie bereits in den Grundlagen (s. Kapitel 2.1) erläutert, kann die Art der Darstellung von Variabilität in Produktmodellen in drei Kategorien (annotativ, kompositional und transformational) eingeteilt werden. Während hierbei für transformationale Methoden mehrere Ansätze existieren, Evolution in Produktmodellen zu modellieren (s. Kapitel 2.4), gibt es für annotative Verfahren, wie etwa Feature-annotierte State Machines, bisher keine Techniken zur Modellierung von Evolution. Die drei Kategorien bieten unterschiedliche Vor- und Nachteile bei der Repräsentation der Modelle und erlauben durch ihre abweichenden Herangehensweisen verschiedene Betrachtungsweisen der Modelle. Je nach gewünschtem Fokus auf bestimmte Eigenschaften der Modelle, wie etwa ein Gesamtüberblick oder Modularität, kann eine geeignete Kategorie gewählt werden. Folglich ist es notwendig, auch für die Modellierung von Evolution in Produktmodellen Ansätze aus verschiedenen Kategorien zur Verfügung zu haben. So kann die Evolution aus unterschiedlichen Blickwinkeln analysiert oder Vergleiche zwischen Methoden verschiedener Kategorien gezogen werden. Daher wird im Folgenden ein annotativer Ansatz zur Modellierung von Evolution vorgestellt. Die bisherigen 150%-Modelle werden dabei auf 175%-Modelle erweitert, sodass sie gleichzeitig Variabilitäts- und Evolutionsinformationen enthalten. Dies ermöglicht, zusätzlich zu einem Überblick über alle Varianten einer Version, gleichzeitig einen Überblick über alle Varianten aller Versionen zu haben.

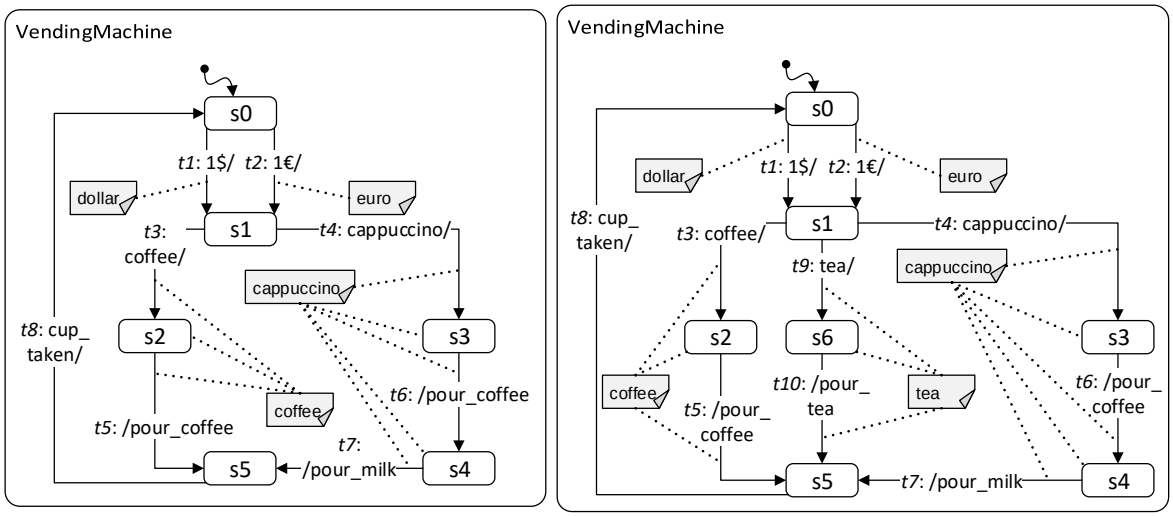
Die Basis für das zu entwickelnde Konzept bilden die Feature-annotierten State Machines von Kamischke et al. [21]. Diese werden derart erweitert, dass die Annotationen an den Elementen ebenfalls die verschiedenen Versionen der Softwareproduktlinie berücksichtigen. Auf diese Weise lässt sich direkt zuordnen, für welche Versionen und Varianten ein bestimmtes Element gültig ist. Die Annotationen in den FaSMs von Kamischke et al. [21] beziehen sich dabei immer auf ein dazugehöriges Feature-Modell. Aus diesem Grund muss auch für die um Versionsinformationen erweiterten FaSMs ein geeignetes Feature-Modell verwendet werden, auf das diese Bezug nehmen können. Theoretisch wäre es hierbei zwar möglich, für jede Version ein eigenes Feature-Modell bereitzustellen, jedoch würde dies bei sehr vielen Versionen zu einer sehr großen und somit unübersichtlichen Menge von Feature-Modellen führen. Daher sollte es sich bei dem zu verwendenden Feature-Modell um ein Modell handeln, dass ebenfalls Evolution darstellen kann.



(a) Feature-Modell zu Zeitpunkt θ_0 (b) Feature-Modell zu Zeitpunkt θ_2

Abbildung 3.1.: Feature-Modell eines Verkaufsautomaten zu unterschiedlichen Zeitpunkten

Zur Veranschaulichung des Konzeptes zur Erweiterung von FaSMs wird die Funktionsweise zunächst beispielhaft dargelegt. Da sich ohne ein eindeutig feststehendes Konzept zur Modellierung von 175%-Modellen nicht zweifelsfrei entscheiden lässt, welche Art von Evolution enthaltendem Feature-Modell zur Kombination am besten geeignet ist, sind in Abbildung 3.1 vorerst zwei einzelne Feature-Modelle für den selben Verkaufsautomaten für verschiedene Versionen dargestellt. Abbildung 3.1a zeigt das Feature-Modell der Version θ_0, während Abbildung 3.1b das Feature-Modell von Version θ_2 zeigt. Bei Version θ_0 bietet die Produktlinie des Verkaufsautomaten die zwei Getränke Kaffee und Cappuccino und fordert die Auswahl einer der zwei Währungen Euro oder Dollar. Ab Version θ_2 wird außerdem zusätzlich Tee als Getränk angeboten. Die entsprechenden Feature-annotierten State Machines, die sich für die jeweilige Version ergeben, sind in Abbildung 3.2 abgebildet. Die beiden State Machines zeigen dabei das gleiche Verhalten, wobei die State Machine in Abbildung 3.2b zusäzlich Elemente zur Auswahl und Ausgabe von Tee enthält.



(a) FaSM zu Zeitpunkt θ_0 (b) FaSM zu Zeitpunkt θ_2

Abbildung 3.2.: Feature-annotierte State Machine eines Verkaufsautomaten zu verschiedenen Zeitpunkten

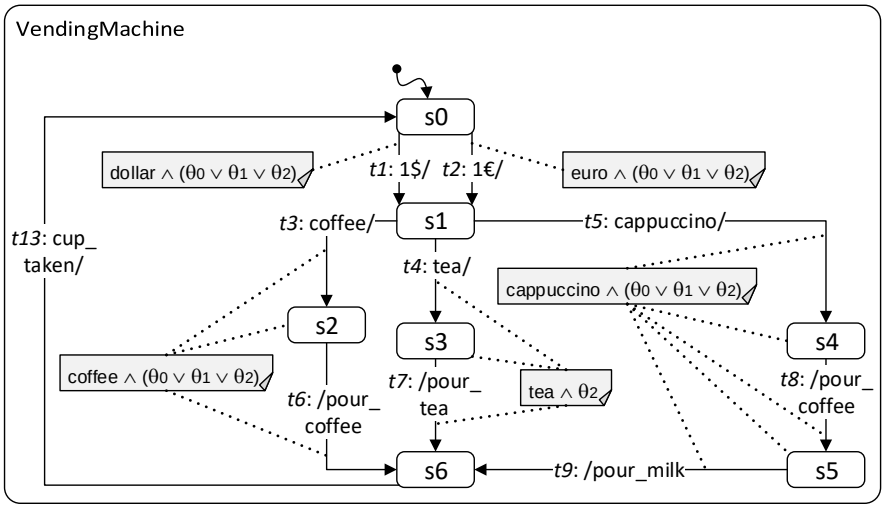


Abbildung 3.3.: Versionierte Feature-annotierte State Machine des Verkaufsautomaten

Um das Verhalten der beiden State Machines innerhalb einer einzigen State Machine repräsentieren zu können, ohne dabei jedoch die Information zu verlieren, welches Element für welche Versionen gültig ist, werden die bisherigen Annotationen erweitert. Jede Auswahlbedingung erhält hierfür zusätzlich eine Konjunktion über die Version, für welche sie gilt. Abbildung 3.3 zeigt eine solche *versionierte Feature-annotierte State Machine* für den Verkaufsautomaten. Da die Elemente `s6`, `t9` und `t10` erst mit Hinzufügen des Features `Tea` ab Version θ_2 existieren, wird dort die neue Auswahlbedingung als Konjunktion aus der alten Bedingung `tea` und der *Versionsbedingung* θ_2 gebildet. Für die restlichen Elemente, welche bereits von Beginn an existieren und daher für alle Versionen gültig sind, bildet sich die Versionsbedingung aus der Disjunktion von θ_0, θ_1 und θ_2.

Auf diese Weise kann die Versionsbedingung für jede mögliche Kombination zusammengesetzt werden. So kann sie etwa als Disjunktion aller Versionen, die betroffen sind, oder auch als Negation der Disjunktion aller Versionen, die nicht betroffen sind, gebildet werden. Die Negation bietet dabei unter Umständen die Möglichkeit, die Versionsbedingung zu kürzen.

Beispiel 1. *Für sechs Versionen θ_0 - θ_5, bei denen `t9` ab θ_2 gültig ist, müsste die Auswahlbedingung lauten: $tea \wedge (\theta_2 \vee \theta_3 \vee \theta_4 \vee \theta_5)$. Mithilfe der Negation kann sie somit auf $tea \wedge \neg(\theta_0 \vee \theta_1)$ verkürzt werden.*

Allerdings ist leicht ersichtlich, dass für lange Evolutionshistorien mit sehr vielen Versionen die Annotationen auf die geschilderte Weise reichlich lang und somit unübersichtlich werden. Des Weiteren ist es aufwändig, jede Version einzeln und explizit in die Annotation aufzunehmen. Daher werden für dieses Konzept Intervalle verwendet, welche die Gültigkeit von Elementen beschreiben. Diese Intervalle sind den Temporalen Validitäten ϑ für Temporale Feature-Modelle von Nieke et al. [31] nachempfunden (s. Kapitel 2.4).

Die Versionsbedingung bildet sich somit aus einem Intervall $\vartheta = [\theta_s, \theta_u)$, welches den Beginn und das Ende der Gültigkeit des Elements definiert. Hierbei muss gelten, dass $\theta_s \in \Theta$, wobei Θ die Menge aller Versionen $\{\theta_0, ..., \theta_n\}$ einer Produktlinie mit $n \geq 0$ ist. Dabei ist θ_0 die erste und θ_n die aktuellste Version der Produktlinie. Der Beginn der Gültigkeit einer Feature-Bedingung ist also immer eine existierende Version der SPL. Da es sich bei ϑ um ein rechtsoffenes Intervall handelt, muss

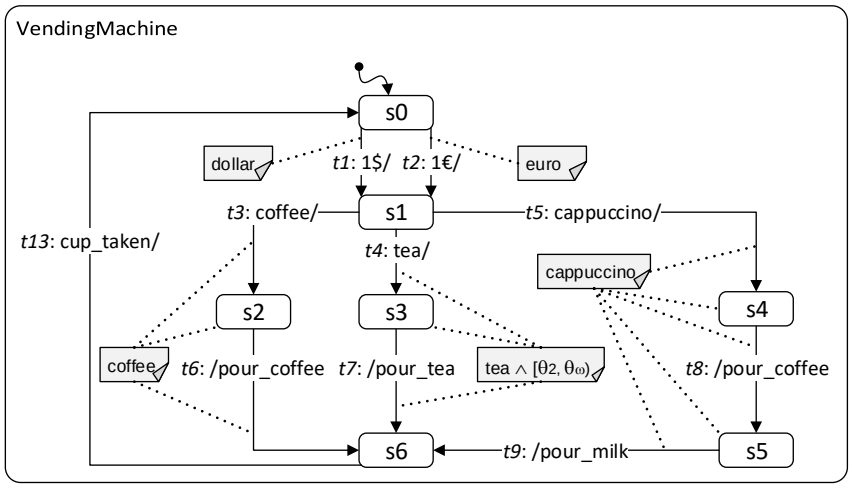
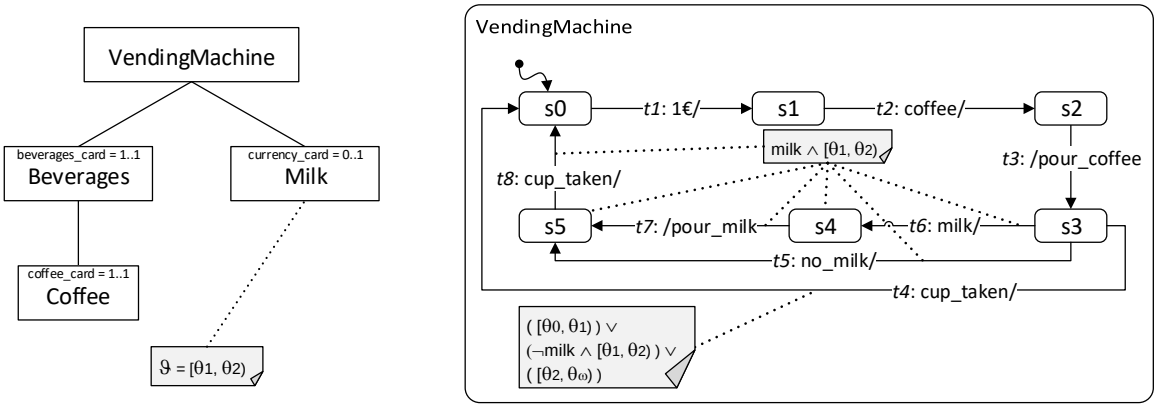


Abbildung 3.4.: Temporale Feature-annotierte State Machine

das Ende der Gültigkeit einer Feature-Bedingung hingegen nicht immer eine bereits existierende Version der SPL sein. Dies würde nämlich bedeuten, dass spätestens für Version θ_n keine einzige Bedingung mehr gültig ist, da θ_u durch das rechtsoffene Intervall nicht im Gültigkeitsbereich liegt. Stattdessen kann θ_u auch auf eine unbekannte, in der Zukunft liegende Version gesetzt werden. Diese unbekannte Version wird hier als θ_ω definiert und bezeichnet eine hypothetische Version, die immer zeitlich später liegt als die momentan aktuellste Version θ_n der Softwareproduktlinie. Daher gilt, dass $\theta_u \in \Theta_\omega$, wobei Θ_ω eine Menge $\{\theta_0, ..., \theta_n, \theta_\omega\}$ mit $0 \leq n < \omega$ ist. Auf diese Weise ist es möglich, mit $\theta_u \mapsto \theta_\omega$ anzugeben, dass eine Feature-Bedingung ab θ_s für jede Version gültig ist. Die Bedingung besitzt somit noch keinen bekannten Zeitpunkt, an dem ihre Gültigkeit endet.

Dieses Konzept ist in Abbildung 3.4 dargestellt. Die Elemente `s6`, `t9` und `t10` besitzen die Annotation $tea \wedge [\theta_2, \theta_\omega)$, da sie erst ab θ_2 gültig sind. Das Ende der Gültigkeit ist auf θ_ω gesetzt, da θ_2 die aktuellste Version ist und somit noch keine Version existiert, ab der die drei Elemente nicht mehr gültig sind. Alle anderen Elemente sind lediglich mit der Feature-Bedingung annotiert. Dies resultiert daher, dass die Versionsbedingung $[\theta_0, \theta_\omega)$ jede mögliche Version der State Machine enthält. Eine Prüfung, ob eine bestimmte Version in diesem Intervall enthalten ist, würde daher immer *true* ergeben. Die Selektionssbedingung für `t1` entspräche somit $dollar \wedge true$. Aus diesem Grund kann auf die zusätzliche Nennung der Versionsbedingung in diesem Fall verzichtet werden. Selbiges Prinzip gilt für die - in jeder Variante enthaltenen - Kernelemente, welche die ganze Lebensdauer der Softwareproduktlinie über gültig bleiben. Diese besitzen theoretisch eine Selektionsbedingung $true \wedge true$, da sie für jegliche Feature-Belegung als auch jede Version gültig sind. Auf Grund dessen wird in diesem Fall die Annotation komplett weggelassen. Für ein Element, das ab einer bestimmten Version θ_i mit $i > 0$ hinzugefügt wird und für jegliche, beliebige Feature-Belegung gilt, also ab diesem Zeitpunkt Kernfunktionen erfüllt, würde sich eine Selektionsbedingung $true \wedge [\theta_i, \theta_\omega)$ ergeben. Hier kann ebenfalls das *true* weggelassen werden, so dass die neue Selektionsbedingung in diesem Fall nur aus der Versionsbedingung besteht.

Da die Annotationen der 175%-Modelle die Versionsbedingung also in Form eines Intervalls entsprechend der Validitäten der Temporalen Feature-Modelle von Nieke et al. [31] enthalten werden,



(a) Temporales Feature-Modell

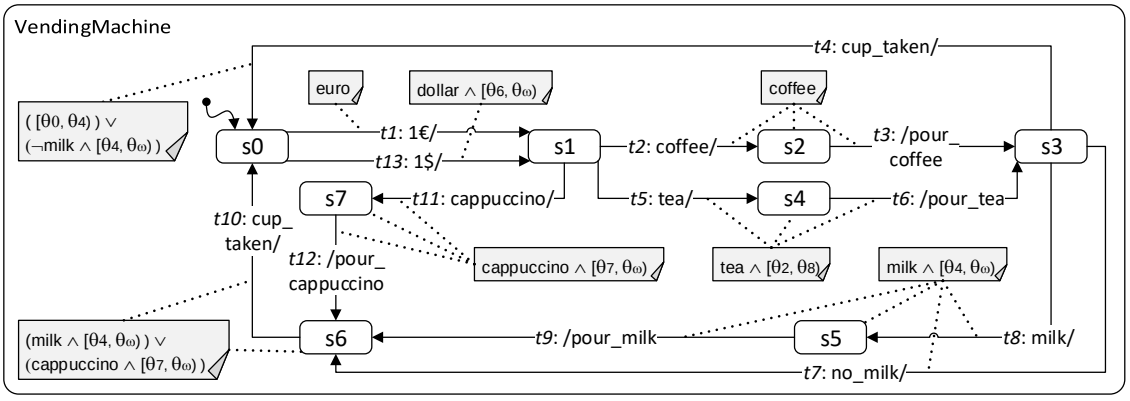
(b) 175%-Modell

Abbildung 3.5.: Konzept für 175%-Modelle am Beispiel eines Verkaufsautomaten

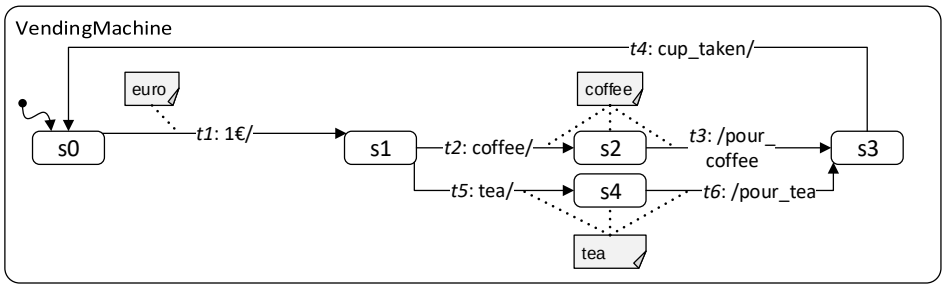
bietet es sich zur Wahrung der Konsistenz an, TFMs als Bezug für die 175%-Modelle zu nehmen. Die erweiterten FaSMs werden daher in Anlehnung nun auch *Temporale Feature-annotierte State Machines* (TSM) genannt.

Modellinhalte einer TSM können im Laufe ihres Lebenszyklus durchaus auch mehrmals verändert werden. So kann ein Element etwa ab einer Version gültig sein, ab einer späteren Version allerdings nicht mehr. Ebenfalls kann ein Element in einer Version für eine (partielle) Konfiguration Γ_1 gültig sein, in einer anderen jedoch für eine Konfiguration Γ_2, wobei $\Gamma_1 \neq \Gamma_2$. Daher kann sich die Selektionsbedingung für ein Element aus einer Disjunktion verschiedener Konjunktionen von Feature-Bedingung und Versionsbedingung zusammensetzen. Hierbei kann sich eine beliebige Anzahl an Disjunktionen ergeben. Abbildung 3.5 verdeutlicht dieses Konzept. Abbildung 3.5a zeigt dabei ein Temporales Feature-Modell eines Verkaufsautomaten, bei dem Feature `Milk` in Version θ_1 hinzugefügt und in Version θ_2 wieder entfernt wird. Darüber hinaus wird immer genau das Getränk `Coffee` angeboten. Für Version θ_1 existieren somit zwei Varianten - eine mit und eine ohne Milch -, während für die Versionen θ_0 und θ_2 lediglich eine Variante existiert. Abbildung 3.5b zeigt das zugehörige 175%-Modell. Für die einfache Variante ohne Milch wird lediglich Geld eingeworfen, Kaffee ausgewählt, eingefüllt und dann der Becher entnommen. Für die Variante mit Milch wird nach Einfüllen des Kaffees ausgewählt, ob Milch gewünscht ist oder nicht, und dann gegebenenfalls Milch eingefüllt. Anschließend wird wieder der Becher entnommen. Die Zustände `s4` und `s5` sowie die Transitionen `t5` bis `t8` sind nur in Version θ_1 für $\Gamma = \{milk \mapsto true\}$ gültig. Transition `t4` hingegen ist in Version θ_0 für jede Konfiguration, in Version θ_1 nur für $\Gamma = \{milk \mapsto false\}$ und in Version θ_2 wiederum für jede Konfiguration gültig. Dementsprechend ergibt sich in diesem Fall als Annotation $[\theta_0, \theta_1) \vee (\neg milk \wedge [\theta_1, \theta_2)) \vee [\theta_2, \theta_\omega)$. Durch die erweiterte Selektionsbedingung kann somit jede beliebige Gültigkeit für ein Element angegeben werden. Gleichzeitig ist die Entwicklung eines Elementes über den kompletten Evolutionszeitraum anhand der Selektionsbedingung ablesbar.

Ein 175%-Modell entwickelt sich mit jedem Evolutionsschritt einer Softwareproduktlinie weiter. Das 175%-Modell für die erste Version einer Produktlinie entspricht dabei einem 150%-Modell, da



(a) Temporale Feature-annotierte State Machine TSM eines Verkaufsautomaten



(b) Versionsfenster TSM_{θ_3} von TSM

Abbildung 3.6.: Konzept für Versionsfenster am Beispiel eines Verkaufsautomaten

zu diesem Zeitpunkt noch keine Evolutionsinformationen vorhanden sind. Genauso wie ein Temporales Feature-Modell die Zusammensammenfassung mehrerer unterschiedlicher Versionen eines Feature-Modells ist (s. Kapitel 2.4), fasst ein 175%-Modell mehrere Versionen von 150%-Modellen zusammen. Indem nur die Elemente betrachtet werden, die für eine bestimmte Version gelten, können daher also ebenfalls 150%-Modelle für jede Version aus dem 175%-Modell extrahiert werden. Dies geschieht auf die gleiche Weise, wie auch Produktmodelle für eine bestimmte Feature-Konfiguration aus 150%-Modellen gewonnen werden. Statt derjenigen Elemente, deren Annotationen eine bestimmte Feature-Konfiguration erfüllen, werden jedoch all die Elemente in das 150%-Modell gewählt, bei denen eine beliebige Versionsbedingung aus der Annotation die gewünschte Version enthält.

Abbildung 3.6 verdeutlicht dieses Prinzip. Abbildung 3.6a zeigt ein 175%-Modell eines Verkaufsautomaten, dessen Produktlinie die Versionen θ_0 bis θ_9 umfasst. Abbildung 3.6b zeigt das resultierende 150%-Modell, das sich ergibt, wenn nur diejenigen Elemente betrachtet werden, die für Version θ_3 unter einer beliebigen Bedingung gültig sind. Diese sind zum einen die Transitionen `t1`, `t2` und `t3` sowie der Zustand `s2`, da diese für jede Version gültig sind, sofern die Bedingungen $coffee$ beziehungsweise *euro* erfüllt sind. Darüber hinaus bleiben die Elemente `t5`, `t6` und `s4` bestehen, da diese unter der Bedingung *tea* für die Versionen θ_2 bis θ_7 gültig sind, als auch das Element `t4`, welches für die Versionen θ_0 bis θ_3 für jede Konfiguration gültig ist. Bei dem so gebildeten 150%-Modell

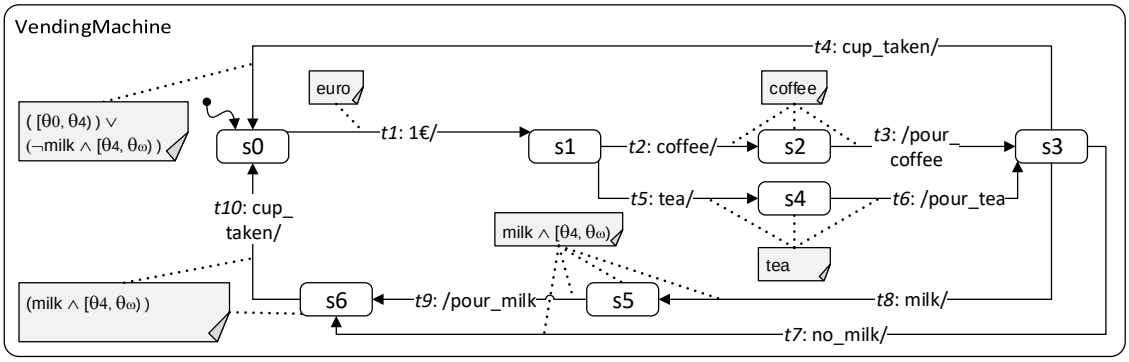


Abbildung 3.7.: Versionsfenster TSM_ϑ von TSM für $\vartheta = [\theta_3, \theta_6)$

handelt es sich um eine Projektion des 175%-Modells, welche nur die Elemente für ein bestimmtes Betrachtungskriterium enthält; in diesem Fall Version θ_3. Da es sich hierbei also um eine Projektion spezifisch für Version θ_3 handelt und somit jedes enthaltene Element für θ_3 gültig ist, wurden die Versionsbedingungen in den Annotationen ausgeblendet. Sie sind somit zwar noch vorhanden, werden in der Projektion der besseren Übersicht halber aber nicht angezeigt. Nach diesem Prinzip entsteht aus einer Temporalen Feature-annotierten State Machine TSM für eine beliebige Version θ das 150%-Modell TSM_θ als Projektion. Dieses Modell ist das *Versionsfenster* von TSM für θ.

Über das Versionsfenster für eine bestimmte Version hinaus, kann durch dieses Vorgehen auch ein Versionsfenster des 175%-Modells für ein Versionsintervall betrachtet werden. In diesem Fall ergibt sich statt eines 150%-Modells wiederum ein 175%-Modell. So können etwa von einem 175%-Modell, das die Versionen θ_0 bis θ_9 umfasst, in einer Projektion all jene Elemente enthalten sein, die für eine beliebige Version im Intervall $\vartheta = [\theta_3, \theta_6)$ gültig sind. Diese Projektion wird als TSM_ϑ bezeichnet und ist das Versionsfenster von TSM für ϑ. Dieses Prinzip wird durch Abbildung 3.7 verdeutlicht. Dort sind alle Elemente von TSM enthalten, die für eine beliebige Konfiguration während des Intervalls $[\theta_3, \theta_6)$ gültig sind. Dies sind zusätzlich zu den Elementen aus Abbildung 3.6b die Elemente `t7`, `t8`, `t9`, `t10`, `s5` und `s6`, da diese unter der Bedingung *milk* ab Version θ_4 gültig sind. Für Transition `t10` wurde hier ein Teil der Annotation ausgeblendet, da $cappuccino \wedge [\theta_7, \theta_\omega)$ außerhalb des betrachteten Intervalls liegt und somit für dieses Versionsfenster keine Relevanz hat. Ebenso wurde wieder für die Elemente `t5`, `t6` und `s4` die Annotation gekürzt, da die Versionsbedingung das betrachtete Intervall komplett einschließt und somit ebenfalls nicht relevant ist.

Das Prinzip des Versionsfensters lässt sich, außer für 175%-Modelle, auch für Temporale Feature-Modelle anwenden. In einem Versionsfenster TFM_θ von einem Temporalen Feature-Modell TFM sind sämtliche Features enthalten, deren Validität die Version θ einschließt, also genau all jene Features, die für Version θ gültig sind. Bei dem resultierenden Modell handelt es sich dementsprechend um ein einfaches Feature-Modell für Version θ, wodurch die Validitäten der Features in der Projektion vernachlässigt werden können. Betrachtet man hingegen das Versionsfenster TFM_ϑ von TFM für ein Versionsintervall ϑ, ergibt sich wiederum ein Temporales Feature-Modell als Projektion. Dieses enthält erneut Features für mehrere Versionen, sodass die Validitäten der enthaltenen Features durchaus relevant sind und nicht ausgeblendet dürfen. Des Weiteren bestehen hier beispielweise auch wieder Gruppen, Kompositionen oder Kardinalitäten als eigene Elemente, die

ebenfalls nicht vernachlässigt werden dürfen, sondern in der Projektion übernommen werden müssen, sofern ihre Validitäten sich mit mindestens einer Version aus dem Intervall ϑ decken. Das Versionsfenster TFM_ϑ enthält also all jene Elemente - nicht nur Features - von TFM, die während des Intervalls ϑ gültig sind.

Nachdem nun die Idee für die Darstellung von Evolution in Feature-annotierten State Machines erläutert wurde, wird im folgenden Kapitel eine formale Definition für die 175%-Modelle gegeben.

3.2. Formale Definition von 175%-Modellen

Die *Temporale Feature-annotierte State Machines* genannten 175%-Modelle dienen dazu, sämtliche Elemente, die während des Lebenszyklus einer SPL jemals in der State Machine eines Produktes auftauchen, in einem Modell zusammenzufassen. Dabei sind alle Elemente mit der Information darüber annotiert, für welche Kombinationen aus Versionen und Feature-Konfiguration sie gültig sind.

$$\Theta = \{\theta_0, \theta_1, ..., \theta_n\} \tag{3.1}$$

bildet die Menge aller existierenden Versionen der Softwareproduktlinie, wobei θ_n die aktuellste Version ist.

Um eine bessere Unterscheidung zwischen 150%- und 175%-Modellen zu gewährleisten, werden die Mengen der Zustände und Transitionen von 150%-Modellen ab sofort als S_{150} und T_{150} und die Gesamtmenge der Elemente somit als $E_{150} = S_{150} \cup T_{150}$ bezeichnet. Existieren mehrere Versionen einer Softwareproduktlinie und demzufolge für jede Version $\theta \in \Theta$ ein eigenes 150%-Modell, werden die Elementmengen der verschiedenen Modelle als

$$E_\theta = S_\theta \cup T_\theta \tag{3.2}$$

unterschieden.

$$E_\Theta = \{E_{\theta_0}, ..., E_{\theta_n}\} \tag{3.3}$$

ist, aus den Gleichungen 3.1 und 3.2 folgend, die Menge, welche die Elementmengen aller 150%-Modelle einer Produktlinie enthält. Ebenso zum Zwecke der Differenzierung wird ein 150%-Modell nun als (M, α_{150}) bezeichnet. (M, α^θ_{150}) ist dementsprechend das 150%-Modell für Version θ, wobei die Annotationsfunktion des Modells als $\alpha^\theta_{150} : E^\theta_{150} \rightarrow \mathbb{B}(F_{FM_\theta})$ definiert ist. FM_θ ist das zugehörige Feature-Modell für die Version θ und F_{FM_θ} folglich die Menge aller Features in FM_θ. Hingegen bestehen Temporale Feature-annotierte State Machines aus einer Menge von Zuständen

$$S_{175} = \bigcup_{i=0}^{n} S_{\theta_i} \tag{3.4}$$

mit einem Startzustand $s_0 \in S_{175}$ und einer Menge von Transitionen

$$T_{175} = \bigcup_{i=0}^{n} T_{\theta_i} \tag{3.5}$$

zwischen den Zuständen. Jeder Zustand und jede Transition, die in einem oder mehreren 150%-Modellen der Softwareproduktlinie auftauchen, sind also genau einmal im 175%-Modell enthalten.

Aus diesem Grund ergibt sich auch genau ein unveränderlicher Startzustand s_0. Würden verschiedene Startzustände für verschiedene Versionen unterstützt werden, könnte es passieren, dass ein Zustand einmal als Startzustand und einmal als normaler Zustand im 175%-Modell vorhanden ist. Dies ist der Fall, wenn der besagte Zustand in einer Version den Startzustand bildet, in einer anderen Version jedoch nur einen normalen Zustand. Um diese Duplizierung von Zuständen zu vermeiden, wird nur ein Startzustand anerkannt.

Transitionen sind beschriftet durch eine Menge von Labeln

$$L_{175} = \bigcup_{i=0}^{n} L_{\theta_i}, \tag{3.6}$$

sodass für jede Transition $t \in T_{175}$ gilt, dass $t = (s, l, s')$. Ein *Label*, oder auch *Transitionsbeschriftung*, $l \in L_{175}$ bildet sich auch hier in der Form $ev[g]/A$, wobei ev ein den Übergang auslösendes Event, g eine Übergangsbedingung, genannt Guard, und A eine Menge von bei Übergang auszuführenden Aktionen bezeichnet. Dementsprechend bildet $E_{175} = \{S_{175} \cup T_{175}\}$ beziehungsweise

$$E_{175} = \bigcup_{i=0}^{n} E_{\theta_i} \tag{3.7}$$

die Menge aller Elemente des Modells. Alle Modellelemente $e \in E_{175}$ der 175%-Modelle besitzen - genau wie die Elemente der 150%-Modelle - Annotationen, welche bezeichnen, für welches Produkt sie angewendet werden. Während sich die Annotationen bei den 150%-Modellen lediglich darauf beziehen, für welche Feature-Konfiguration ein Element gilt, kennzeichnen die Annotationen der 175%-Modelle sowohl für welche Feature-Konfigurationen als auch für welche Versionen der Produktlinie ein Element verwendet wird. Wie in Abschnitt 3.1 erläutert, kann sich eine Annotation dabei aus mehreren aussagenlogischen Formeln über Feature-Parameter zusammensetzen, die sich auf je eine Version oder ein Versionsintervall beziehen. Durch eine Funktion

$$\varphi : E_{175} \to \mathcal{P}(\Theta) \tag{3.8}$$

wird jedem Element $e \in E_{175}$ eine Menge von Versionen θ zugeordnet. Hierbei gilt, dass

$$\forall e \in E_{175} : \varphi(e) = \{\theta \in \Theta \mid \exists E_\theta \in E_\Theta : e \in E_\theta\}. \tag{3.9}$$

Für jedes $e \in E_{175}$ enthält $\varphi(e)$ also alle diejenigen Versionen $\theta \in \Theta$, in deren Elementmenge E_θ das Element e enthalten ist.

Jeder Version $\theta \in \varphi(e)$ wird darüber hinaus eine aussagenlogische Formel über Feature-Parameter in F_{TFM} zugeordnet.

$$F_{TFM} = \{f_1, ..., f_n\} \tag{3.10}$$

ist dabei die Menge aller Features des Temporalen Feature-Modells TFM, welches den Bezugspunkt für das 175%-Modell bildet. Die Menge aller aussagenlogischen Formeln, die sich über die Features von TFM bilden lassen, wird als $\mathbb{B}(F_{TFM})$ bezeichnet. Die aussagenlogischen Formeln werden den Versionen durch die Funktion

$$\alpha_{175} : E_{175} \times \mathcal{P}(\Theta) \to \mathcal{P}(\Theta \times \mathbb{B}(F_{TFM})) \tag{3.11}$$

zugeordnet. Auf diese Weise wird jedem Tupel $(e, \varphi(e))$ eine Menge $\alpha_{175}(e, \varphi(e))$ von Tupeln (θ, γ) mit $\gamma \in \mathbb{B}(F_{TFM})$ zugewiesen. Das Tupel $(e, \varphi(e))$ beschreibt hierbei das Element e und die Menge der ihm zugeordneten Versionen θ. Die Menge $\alpha_{175}(e, \varphi(e))$ enthält alle Kombinationen aus Versionen θ und der für die Versionen jeweils gültigen aussagenlogischen Formel γ in Form von Tupeln (θ, γ). An dieser Stelle gilt, dass

$$\forall e \in E_{175} : \alpha_{175}(e, \varphi(e)) = \{(\theta, \gamma) \in \Theta \times \mathbb{B}(F_{TFM}) \mid \theta \in \varphi(e), \exists (M, \alpha^{\theta}_{150}) : \gamma = \alpha^{\theta}_{150}(e)\}. \quad (3.12)$$

Die Menge $\alpha_{175}(e, \varphi(e))$ umfasst also genau jene Tupel (θ, γ), für die zum einen gilt, dass θ in der Menge der e zugeordneten Versionen $\varphi(e)$ enthalten ist. Zum anderen muss gelten, dass die aussagenlogische Formel γ genau der Annotation $\alpha^{\theta}_{150}(e)$ entspricht, welche dem Element e im 150%-Modell von Version θ zugewiesen wurde.

Die Elemente der 175%-Modelle werden somit durch α_{175} mit der Information darüber annotiert, für welche Produktmodelle sie gültig sind. Dabei können die Elemente für unterschiedliche Versionen gültig sein und innerhalb dieser Versionen wiederum für unterschiedliche Feature-Konfigurationen. Ein Element e gehört genau dann zu einem bestimmten Produktmodell, wenn mindestens eines der Tupel $(\theta, \gamma) \in \alpha_{175}(e, \varphi(e))$ für das Produktmodell erfüllt ist. Ein Tupel ist für ein Produktmodell immer dann erfüllt, wenn sowohl θ der Version entspricht, auf die sich das Produktmodell bezieht, als auch γ der Feature-Konfiguration entspricht, durch die das Produktmodell definiert wird. Zusammenfassend sind 175%-Modelle somit wie folgt definiert:

Definition 1. *Eine Temporale Feature-annotierte State Machine oder ein 175%-Modell ist ein Tupel* $(M, \alpha_{175}) = (S_{175}, s_0, T_{175}, L_{175}, \varphi, \alpha_{175})$, *wobei*

- S_{175} *eine Menge von Zuständen* s,
- $s_0 \in S_{175}$ *ein Startzustand,*
- T_{175} *eine Menge von Transitionen* $t = (s, l, s')$,
- L_{175} *eine Menge von Transitionsbeschriftungen (Label)* l,
- φ *eine Versionsfunktion* $\varphi : E_{175} \to \mathcal{P}(\Theta)$,
- *und* α_{175} *eine Konfigurationsfunktion* $\alpha_{175} : E_{175} \times \mathcal{P}(\Theta) \to \mathcal{P}(\Theta \times \mathbb{B}(F_{TFM}))$ *ist.*

Ein 175%-Modell wird, Bezug nehmend auf die Definition von α_{175} und im Einklang mit den 150%-Modellen, als (M, α_{175}) bezeichnet. Folglich bezeichnet $(M, \alpha_{175})_{\theta}$, wie in Kapitel 3.1 beschrieben, nun ein Versionsfenster von (M, α_{175}) für Version θ. Das Versionsfenster ist eine Projektion des 175%-Modells, welches alle Elemente e enthält, für die das erste Argument eines beliebigen Tupels $(\theta, \gamma) \in \alpha_{175}(e, \varphi(e))$ mit der Version θ übereinstimmt. $(M, \alpha_{175})_{\theta}$ bildet somit ein 150%-Modell für Version θ, wodurch seine Inhalte mit dem 150%-Modell $(M, \alpha^{\theta}_{150})$ übereinstimmen. Sowohl $(M, \alpha_{175})_{\theta}$ als auch $(M, \alpha^{\theta}_{150})$ repräsentieren somit das gleiche Modell und unterscheiden sich ausschließlich darin, dass $(M, \alpha^{\theta}_{150})$ ein eigenständiges Modell bildet, während $(M, \alpha_{175})_{\theta}$ lediglich eine Projektion des 175%-Modells darstellt.

Damit es bei den Annotationen der 175%-Modelle nicht zu Inkonsistenzen kommt, muss für jedes Tupel (θ, γ) gelten, dass $\gamma \vDash TFM_{\theta}$. Die aussagenlogische Formel γ muss also die Einschränkungen des Versionsfensters TFM_{θ} vom Temporalen Feature-Modell TFM für das Intervall θ erfüllen.

Auch hier gilt, dass TFM_θ und FM_θ das gleiche Feature-Modell respräsentieren, wobei TFM_θ nur eine Projektion des Temporalen Feature-Modells für die Version θ ist, während FM_θ ein eigenständiges Modell verkörpert. Daher kann gleichermaßen auch $\gamma \vDash FM_\theta$ gelten, sollte kein Temporales Feature-Modell als Bezugspunkt zur Verfügung stehen, sondern nur einzelne Feature-Modelle.

Wird eine Softwareproduktlinie um eine neue Version erweitert, dann verändert sich sowohl das Temporale Feature-Modell als auch das 175%-Modell. In beiden Fällen werden keine Modellinhalte gelöscht, selbst wenn in der neuen Version der SPL im Vergleich zu vorher ein Feature entfernt wird. Das Feature beziehungsweise die Modellinhalte, die von diesem Feature abhängig sind, bleiben stattdessen im TFM und im 175%-Modell enthalten und werden lediglich invalidiert. Das bedeutet, dass die Validität des Features und die Annotationen der 175%-Modellelemente so angepasst werden, dass sie angeben, dass Feature und Elemente ab der neuen Version nicht mehr gültig sind. In einem Temporalen Feature-Modell sowie in einem 175%-Modell sind also immer alle Elemente enthalten, die jemals während des Lebenszyklus der Softwareproduktlinie für eine beliebige Version existiert haben.

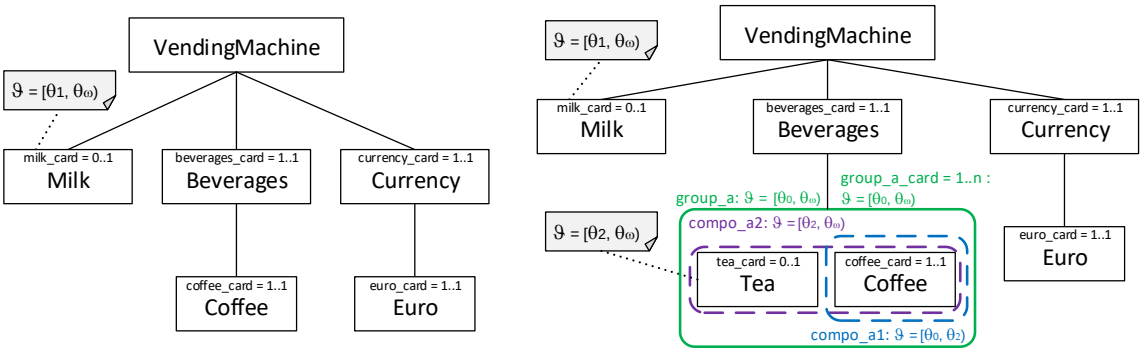
Bei jedem Evolutionsschritt, der eine neue Version θ_{n+1} hinzufügt, wird faktisch das 150%-Modell der bisher aktuellsten Version θ_n als Referenz herangezogen. Das 150%-Modell von Version θ_{n+1} wird dann erstellt, indem das vorherige Modell an die neuen Anforderungen der Produktlinie angepasst wird. Es wird also das Modell $(M, \alpha_{175})_{\theta_n}$ verwendet und gegebenenfalls Elemente hinzugefügt, entfernt oder modifiziert, sodass sich das 150%-Modell $(M, \alpha_{150}^{\theta_{n+1}})$ für die neue Version θ_{n+1} ergibt. Das erweiterte 175%-Modell (M, α_{175}) entsteht dann, indem zunächst die Elementmenge des Modells aktualisiert wird. Für die neue Elementmenge E_{175} gilt, dass

$$E_{175} = E_{175} \cup E_{\theta_{n+1}}. \tag{3.13}$$

Die bisherigen Elemente des 175%-Modells werden also mit den Elementen des neuen 150%-Modells $(M, \alpha_{150}^{\theta_{n+1}})$ vereinigt, und das 175%-Modell somit vergrößert. Ebenso wird die Menge Θ der existierenden Versionen durch das Ergänzen der neuen Version θ_{n+1} aufgestockt. Da dadurch nun eine Version mehr für die Produktlinie zur Verfügung steht, besteht die Möglichkeit, dass sich die Mengen $\varphi(e)$ der bisherigen Elemente $e \in E_{175}$ verändern. Dies hat wiederum Auswirkungen auf $\alpha_{175}(e, \varphi(e))$, weshalb die Annotationen der bisherigen Elemente e aus E_{175} angepasst werden müssen. Die neu hinzugekommenen Elemente erhalten darüber hinaus ebenfalls Annotationen. Folgende Änderungen werden an den Annotationen der Elemente vorgenommen:

- Für Elemente e, die in $(M, \alpha_{175})_{\theta_n}$ enthalten waren, in $(M, \alpha_{150}^{\theta_{n+1}})$ aber nicht mehr enthalten sind, also entfernt wurden, ergibt sich keine Änderung an der Annotation. Dies liegt daran, dass sich ihre Menge $\varphi(e)$ nicht verändert, da sie in der neuen Version θ_{n+1} nicht auftauchen.
- Elemente e, die in $(M, \alpha_{175})_{\theta_n}$ und auch in (M, α_{175}) nicht enthalten waren, aber in $(M, \alpha_{150}^{\theta_{n+1}})$ enthalten sind, also zum ersten Mal in der Historie hinzugefügt wurden, erhalten die Versionsmenge $\varphi(e) = \{\theta_{n+1}\}$ und als Annotation $\alpha_{175}(e, \varphi(e)) = (\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e))$.
- Hingegen wird für Elemente e, die in $(M, \alpha_{175})_{\theta_n}$ nicht enthalten waren, aber in $(M, \alpha_{150}^{\theta_{n+1}})$ enthalten sind und auch bereits in (M, α_{175}) enthalten waren, also nicht zum ersten Mal in der Historie hinzugefügt wurden, die Versionsmenge $\varphi(e)$ erweitert, sodass $\varphi(e) = \varphi(e) \cup \{\theta_{n+1}\}$, und außerdem der Annotation ein weiteres Tupel $(\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e))$ hinzugefügt, sodass $\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e)) \cup (\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e))$.

- Für Elemente e, die in $(M, \alpha_{175})_{\theta_n}$ enthalten waren und in $(M, \alpha_{150}^{\theta_{n+1}})$ ebenfalls enthalten sind, an deren Existenz sich also nichts ändert, erfolgt eine Erweiterung von $\varphi(e)$ und somit auch von $\alpha_{175}(e, \varphi(e))$. Dabei wird $\varphi(e)$ ergänzt durch θ_{n+1}, sodass $\varphi(e) = \varphi(e) \cup \theta_{n+1}$. Dementsprechend wird auch zu $\alpha_{175}(e, \varphi(e))$ ein neues Tupel $(\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e))$ hinzugefügt, sodass $\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e)) \cup (\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e))$.
- Bei Elementen e, die in $(M, \alpha_{175})_{\theta_n}$ enthalten waren und in $(M, \alpha_{150}^{\theta_{n+1}})$ in einer veränderten Form enthalten sind, also modifiziert wurden, wird die neue Form als Kopie e' behandelt. Da e' somit ein neues Element darstellt, wird e' wie ein hinzugefügtes Element gehandhabt, während mit e wie mit einem entfernten Element verfahren wird. Für e ergibt sich somit keine Änderung der Annotation. Wenn e' noch nicht in (M, α_{175}) vorhanden ist, also vorher noch nie existiert hat, erhält e' folglich die Versionsmenge $\varphi(e') = \{\theta_{n+1}\}$ und die Annotation $\alpha_{175}(e', \varphi(e')) = (\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e'))$. Sollte e' hingegen schon in (M, α_{175}) enthalten sein, also bereits zu einem früheren Zeitpunkt existiert haben, wird die Versionsmenge folglich auf $\varphi(e') = \varphi(e') \cup \{\theta_{n+1}\}$ erweitert und ein neues Tupel $(\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e'))$ an die bestehende Annotation angehängt, sodass $\alpha_{175}(e', \varphi(e')) = \alpha_{175}(e', \varphi(e')) \cup (\theta_{n+1}, \alpha_{150}^{\theta_{n+1}}(e'))$ gilt.



(a) Temporales Feature-Modell für $\theta_n = \theta_1$

(b) Temporales Feature-Modell für $\theta_n = \theta_2$

Abbildung 3.8.: Temporales Feature-Modell vor und nach Erweiterung

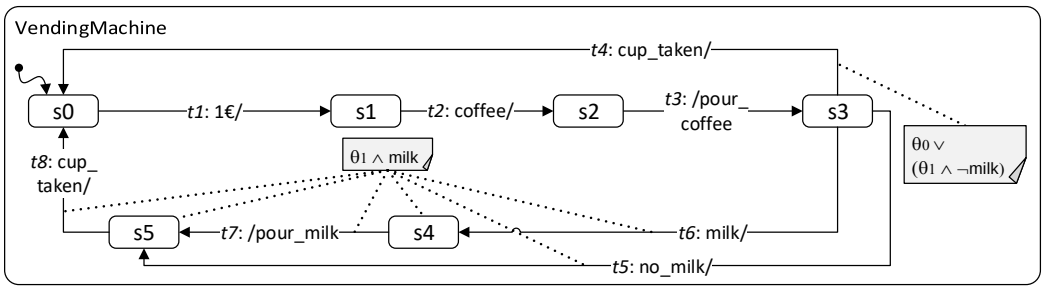
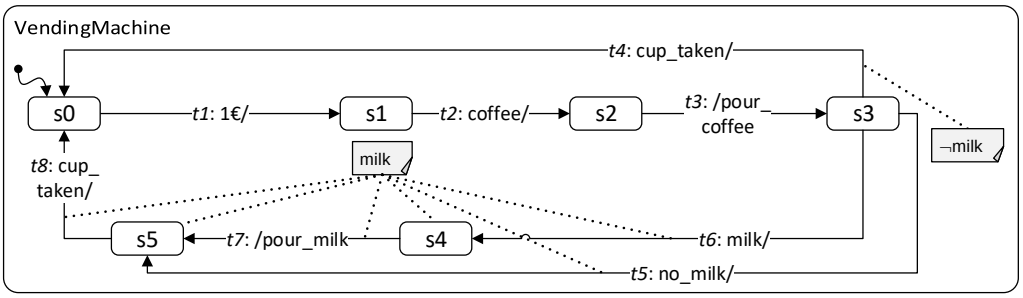


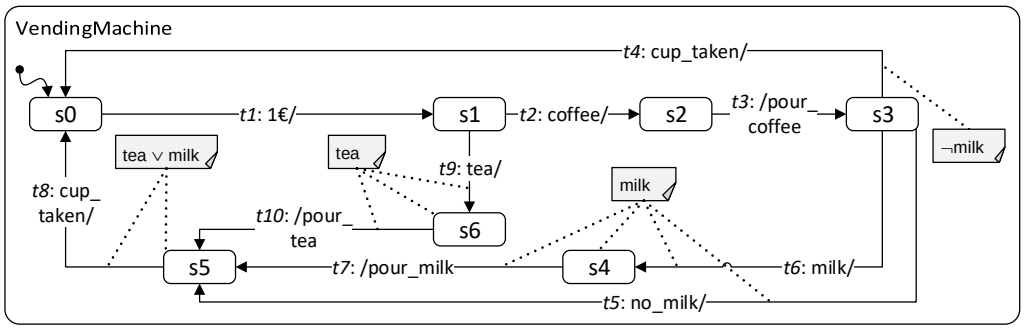
Abbildung 3.9.: 175%-Modell (M, α_{175}) für $\theta_n = \theta_1$

Auf die geschilderte Weise können 175%-Modelle für jeden neuen Evolutionsschritt angepasst und erweitert werden. Im Folgenden wird anhand eines Beispiels die Verwendung der soeben vor-

gestellten Definition demonstriert. Die Produktlinie des Beispielverkaufsautomaten besteht zunächst aus den Versionen θ_0 und θ_1 und soll um eine neue Version θ_2 erweitert werden. Dazu zeigt zunächst Abbildung 3.8 zwei Temporale Feature-Modelle für den Verkaufsautomaten, einmal mit θ_1 als aktuellster Version und einmal mit θ_2 als aktuellster Version. Für Version θ_0 sind lediglich die Features `Beverages` und `Currency` mit ihren Kind-Features `Coffee` und `Euro` vorhanden. Ab Version θ_1 kommt Feature `Milk` hinzu und in Version θ_2 folgt Feature `Tea`. In Abbildung 3.9 ist das 175%-Modell dargestellt, welches sich für θ_1 als aktuellste Version ergibt.



(a) 150%-Modell $(M, \alpha_{150}^{\theta_1})$



(b) 150%-Modell $(M, \alpha_{150}^{\theta_2})$

Abbildung 3.10.: 150%-Modelle für θ_1 und θ_2

Abbildung 3.10a zeigt darüber hinaus das 150%-Modell für die Version θ_1, und somit das aktuellste 150%-Modell der Produktlinie. Abbildung 3.10b enthält das 150%-Modell, welches sich nach Anwendung des Evolutionsschritts ergeben soll, also das Modell für die neue Version θ_2. Für Version θ_1 lautet die Menge aller existierenden Versionen daher zunächst noch $\Theta = \{\theta_0, \theta_1\}$. Die Menge $E_{\theta_1} = \{s0, s1, s2, s3, s4, s5, t1, t2, t3, t4, t5, t6, t7, t8\}$ aller Elemente von $(M, \alpha_{150}^{\theta_1})$ entspricht gleichzeitig der aktuellen Menge E_{175} von (M, α_{175}) aus Abbildung 3.9. Für das neue 150%-Modell für Version θ_2 ist $E_{\theta_2} = \{s0, s1, s2, s3, s4, s5, s6, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10\}$ die Menge aller Elemente des Modells.

Für die Beschreibung der Annotationen werden hier nur die Elemente `s0`, `s5`, `t4` und `t5` betrachtet, da die restlichen Elemente sich die Annotation mit einem dieser Elemente teilen. Den

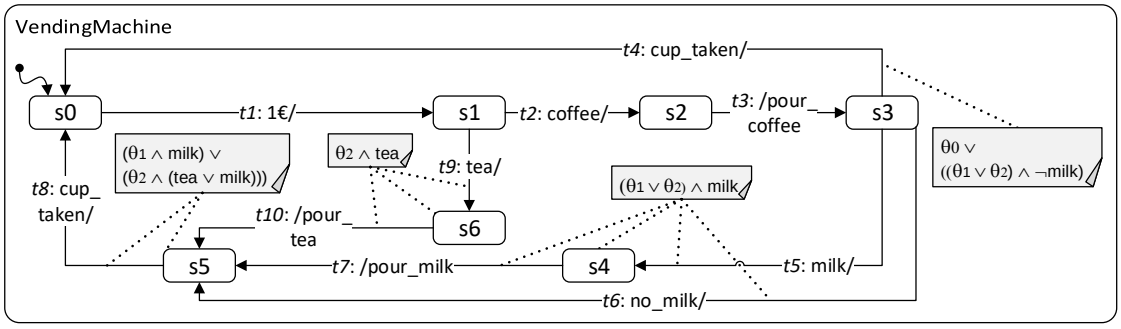


Abbildung 3.11.: 175%-Modell (M, α_{175})

Elementen `s0` und `t4` werden durch die Funktion φ folgende Versionen zugeordnet:

$$\varphi(s0) = \varphi(t4) = \{\theta_0, \theta_1\}.$$

Für die Elemente `s5` und `t5` hingegen gilt:

$$\varphi(s5) = \varphi(t5) = \{\theta_1\}.$$

Als Annotationen ergeben sich daher für die vier Elemente folgende Tupel:

$$\begin{aligned}
\alpha_{175}(s0, \{\theta_0, \theta_1\}) &= \{(\theta_0, true), (\theta_1, true)\} \\
\alpha_{175}(s5, \{\theta_1\}) &= \{(\theta_1, milk)\} \\
\alpha_{175}(t4, \{\theta_0, \theta_1\}) &= \{(\theta_0, true), (\theta_1, \neg milk)\} \\
\alpha_{175}(t5, \{\theta_1\}) &= \{(\theta_1, milk)\}
\end{aligned}$$

Jetzt wird Θ um die neue Version θ_2 ergänzt, sodass gilt $\Theta = \{\theta_0, \theta_1, \theta_2\}$. Um nun ein erweitertes 175%-Modell zu erhalten wird die alte Menge E_{175} mit der Menge E_{θ_2} vereinigt wodurch sich als neue Menge $E_{175} = \{s0, s1, s2, s3, s4, s5, s6, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10\}$ ergibt. Das neue 175%-Modell, das sich unter Einbezug der neuen Version θ_2 ergibt und in Abbildung 3.11 dargestellt ist, besteht daher nun aus:

- $S_{175} = \{s0, s1, s2, s3, s4, s5, s6\}$
- $s_0 = s0$
- $T_{175} = \{t1, t2, t3, t4, t5, t6, t7, t8, t9, t10\}$
- $L_{175} = \{1€/,\ coffee/,\ /pour\_coffee,\ cup\_taken/,\ no\_milk/,\ milk/,\ /pour\_milk\}$

Für die Beschreibung der durch die Erweiterung des Modells folgende Veränderung der Annotationen wird zusätzlich zu den Elementen `s0`, `s5`, `t4` und `t5` auch Transition `t9` betrachtet, da es sich hierbei um ein neu hinzugekommmenes Element handelt. Die restlichen, neu hinzugefügten Elemente werden nicht extra betrachtet, da sie sich eine Annotation mit `t9` teilen. Die Versionsmengen von `s0`, `s5`, `t4` und `t5` werden um die neue Version θ_2 erweitert, sodass nun gilt

$$\varphi(s0) = \varphi(t4) = \{\theta_0, \theta_1\} \cup \theta_2 = \{\theta_0, \theta_1, \theta_2\}$$

und

$$\varphi(s5) = \varphi(t5) = \{\theta_1\} \cup \theta_2 = \{\theta_1, \theta_2\}.$$

Element `t9` erhält die neue Versionsmenge

$$\varphi(t9) = \{\theta_2\}.$$

Für die neue Version erhält dementsprechend auch jedes der Elemente ein neues Tupel für die Annotation. Die Tupel bestehen dabei aus der Version θ_2 und aus der Selektionsbedingung des jeweiligen Elementes für $(M, \alpha_{150}^{\theta_2})$. Die Selektionsbedingung können Abbildung 3.10b entnommen werden. Somit ergeben sich folgende Annotationen für die Elemente:

$$\begin{aligned}
\alpha_{175}(s0, \{\theta_0, \theta_1, \theta_2\}) &= \{(\theta_0, true), (\theta_1, true), (\theta_2, true)\} \\
\alpha_{175}(s5, \{\theta_1, \theta_2\}) &= \{(\theta_1, milk), (\theta_2, tea \vee milk)\} \\
\alpha_{175}(t4, \{\theta_0, \theta_1, \theta_2\}) &= \{(\theta_0, true), (\theta_1, \neg milk), (\theta_2, \neg milk)\} \\
\alpha_{175}(t5, \{\theta_1, \theta_2\}) &= \{(\theta_1, milk), (\theta_2, milk)\} \\
\alpha_{175}(t9, \{\theta_2\}) &= \{(\theta_2, tea)\}
\end{aligned}$$

Bei der grafischen Darstellung von 175%-Modellen können Tupel mit aufeinanderfolgenden Versionen der Übersichtlichkeit halber zusammengefasst werden, sofern den Versionen durch das Tupel die gleiche aussagenlogische Formel zugeordnet ist. So können beispielsweise für Element `s0` die Tupel $(\theta_0, true)$, $(\theta_1, true)$ und $(\theta_2, true)$ zu einem Tupel $(\{\theta_0, \theta_1, \theta_2\}, true)$ mit expliziter Formulierung der Versionsmenge vereint werden. Eine andere Möglichkeit besteht darin, die Versionsmenge durch ein Intervall in einem Tupel $([\theta_0, \theta_2], true)$ zu repräsentieren.

Die verschiedenen Tupel der Annotation lassen sich des Weiteren in einer aussagenlogischen Formel darstellen. Wie bereits in der Definition erläutert, ist ein Element in einem bestimmten Produktmodell enthalten, wenn mindestens eines der Tupel aus der Annotation durch die Eigenschaften des Produktmodells (Version und Feature-Konfiguration) erfüllt ist. Dieser Zusammenhang entspricht in der Aussagenlogik einer Oder-Verknüpfung, sodass sich beispielsweise die drei Tupel von `s0` als Formel $(\theta_0, true) \vee (\theta_1, true) \vee (\theta_2, true)$ bilden lassen. Diese Formel lässt sich nach dem Kommutativgesetz wiederum zu einem Tupel $((\theta_0 \vee \theta_1 \vee \theta_2), true)$ zusammenfassen. Auch die einzelnen Tupel selbst können in aussagenlogischer Form repräsentiert werden. Ein Tupel einer Annotation ist für ein bestimmtes Produktmodell genau dann erfüllt, wenn beide Argumente des Tupels den Eigenschaften des Produktmodells entsprechen. Diese Verbindung wird in der Aussagenlogik durch eine Und-Verknüpfung wiedergegeben. Daher kann ein Tupel $(\theta_0, true)$ als Formel $\theta_0 \wedge true$ dargestellt werden. Zusammen ergibt sich somit für die Annotation von `s0` die aussagenlogische Formel $(\theta_0 \wedge true) \vee (\theta_1 \wedge true) \vee (\theta_2 \wedge true)$ oder in verkürzter Form $(\theta_0 \vee \theta_1 \vee \theta_2) \wedge true$.

Auf diese Weise ergeben sich auch die Annotationen der Elemente in Abbildung 3.11. Für Element `s5` wird $\{(\theta_1, milk), (\theta_2, tea \vee milk)\}$ dargestellt als $(\theta_1 \wedge milk) \vee (\theta_2 \wedge (tea \vee milk))$ und für Element `t4` wird $\{(\theta_0, true), (\theta_1, \neg milk), (\theta_2, \neg milk)\}$ repräsentiert als $(\theta_0 \wedge true) \vee ((\theta_1 \vee \theta_2) \wedge \neg milk)$. In einer Und-Verknüpfung besteht die Möglichkeit ein *true* wegzulassen, da der Wahrheitswert der kompletten Formel in diesem Fall nur noch auf den Wahrheitswerten der restlichen Teilaussagen basiert. Somit lässt sich die Formel für `t4` auf $\theta_0 \vee ((\theta_1 \vee \theta_2) \wedge \neg milk)$ verkürzen. Des Weiteren lässt

sich $\{(\theta_1, milk), (\theta_2, milk)\}$ für `t5` als $(\theta_1 \vee \theta_2) \wedge milk$ formulieren, während für `t9` aus $\{(\theta_2, tea)\}$ die Formel $\theta_2 \wedge tea$ gebildet wird. Für Element `s0` ist in der Abbildung keine Annotation dargestellt, da die Formel $(\theta_0 \vee \theta_1 \vee \theta_2) \wedge true$ für jede mögliche Belegung in der beispielhaften Produktlinie erfüllt ist und sich als Wahrheitswert der Formel somit immer *true* ergibt. Aus diesem Grund kann die Annotation in der Darstellung weggelassen werden.

Temporale Feature-annotierte State Machines bieten somit einen Gesamtüberblick über jegliches abzuleitende Produktmodell einer Softwareproduktlinie. Allerdings werden auch diese 175%-Modelle bei variantenreichen Produktlinien mit einer langen Evolutionshistorie schnell sehr groß und somit unübersichtlich. Aus diesem Grund und damit auch auf 175%-Modellen Analysen für lediglich bestimmte Aspekte des Modells durchgeführt werden können, wird im folgenden Kapitel ein Slicing-Algorithmus für Temporale Feature-annotierte State Machines vorgestellt.

3.3. Slicing von 175%-Modellen

Das Slicing von Temporalen Feature-annotierten State Machines soll ermöglichen, die Modelle auf die Modellinhalte zu reduzieren, die für ein bestimmtes Betrachtungskriterium C_{175} relevant sind. Das Kriterium $C_{175} = (e, \Gamma, \vartheta)$ setzt sich zusammen aus einem Element $e \in E_{175}$, einer (partiellen) Feature-Konfiguration Γ und einem abgeschlossenen Intervall $\vartheta = [\theta_i, \theta_j]$ von Versionen θ_k mit $0 \leq k \leq n$ und $i \leq j$, wobei θ_n die aktuellste Version der Softwareproduktlinie ist.

Das Slicing von 175%-Modellen folgt hierbei dem gleichen Vorgehen wie das Slicing von Feature-annotierten State Machines (s. Kapitel 2.3). Dieses wird allerdings dahingehend erweitert, dass nun zusätzlich zu der Berücksichtigung der Feature-Bedingungen auch die Versionsbedingungen einbezogen werden. Ein 175%-Modell kann dabei sowohl für eine Kombination aus Versionsintervall ϑ und Feature-Konfiguration Γ als auch nur für ein Intervall oder nur für eine Konfiguration reduziert werden. Es müssen also nicht alle Werte des Kriteriums C_{175} definiert werden. Wird nur Γ definiert, dann ergibt sich ein Slice, der alle Elemente enthält, deren Annotation Γ erfüllt, unabhängig davon für welche Versionen das Element gilt. Wenn nur ϑ definiert wird, ergibt sich umgekehrt ein Slice, der all diejenigen Elemente enthält, die für ein bestimmtes Intervall ϑ gültig sind, ungeachtet dessen, für welche Feature-Konfigurationen sie gelten. Durch ϑ kann hierbei eine einzige Version oder mehrere zeitlich aufeinander folgende Versionen für das Kriterium spezifiziert werden. Ein Slice, bei dem nur ϑ für das Kriterium definiert wird, ergibt das Versionsfenster für ϑ von der Temporalen Feature-annotierten State Machine.

Algorithmus 3.1 .: 175%-State Machine Slicing Algorithmus

Input : 175%-State Machine $M_{175} = (M, \alpha_{175})$, Slicing-Kriterium $C_{175} = (e, \Gamma, \vartheta)$
Output : Slice $M_{C_{175}}$

1 $Dep_M := computeDep(M_{175})$;
2 $M_0 := initSlice(M_{175}, C_{175})$;
3 **repeat**
4 $\quad M_{i+1} = reachable(M'_i, Dep_M, \pi, \Gamma)$;
5 $\quad M'_{i+1} = wellformed^{(+)}(M_{i+1})$;
6 **until** $M'_{i+1} = M'_i$;
7 $M_{C_{175}} := wellformed^{(-)}(M'_{i+1})$;

Algorithmus 3.1 beschreibt das Vorgehen beim Slicing von 175%-Modellen. Die Eingabe für diesen Algorithmus besteht aus einer Feature-annotierten State Machine $M_{175} = (M, \alpha_{175})$ und einem bedingten Slicing-Kriterium $C_{175} = (e, \Gamma, \vartheta)$. Als Ausgabe ergibt sich ein Slice $M_{C_{175}}$ von M_{175} für C_{175}. Wie zuvor bei den 150%-Modellen werden auch hier zunächst die Abhängigkeiten zwischen den Elementen des 175%-Modells erstellt. Danach folgt ebenso wie zuvor die Erstellung des initialen Slices, der zu Beginn lediglich Element e des Slicing-Kriteriums C_{175} enthält. Im Anschluss daran werden iterativ alle Elemente e' zum Slice hinzugefügt, wenn eine Abhängigkeit $\rightarrow_{dep} \in Dep_M$ besteht, sodass $e \rightarrow_{dep} e'$ und $e \in M_i'$. Es werden also alle Elemente hinzugefügt, von denen ein beliebiges Element, das bereits im Slice vorhanden ist, abhängig ist. Hierbei gilt für 175%-Modelle nun, dass dabei nur jene Elemente e' zum Slice hinzugefügt werden, die sowohl das Intervall ϑ als auch die Konfiguration Γ des Kriteriums C_{175} durch ihre Selektionsbedingung erfüllen. Sobald eine der beiden Kriteriumsbedingungen nicht vom Element e' erfüllt wird, wird das Element nicht in den Slice mit aufgenommen. Bei jeder Iteration erfolgt ferner die Aufnahme weiterer Elemente in den Slice, die für die Erstellung einer wohlgeformten State Machine benötigt werden. Hierbei muss die Erfüllung der Kriteriumsbedingungen nicht weiter kontrolliert werden, da wie bei den 150%-Modellen ein Erstellen einer wohlgeformten State Machine immer möglich ist, indem lediglich Elemente zum Slice hinzugefügt werden, welche ϑ und Γ erfüllen. Sobald der Fixpunkt erreicht ist, das heißt, sobald keine weiteren Elemente e' mehr zum Slice hinzugefügt werden, wird ein letztes Mal die Wohlgeformheit des Modells überprüft und alle Elemente entfernt, welche diese nicht erfüllen. Auch hierbei werden die Bedingungen des Kriteriums nicht mit einbezogen, da beim Entfernen von Elementen, welche nicht der Wohlgeformtheit entsprechen, die Annotationen der Elemente nicht relevant sind.

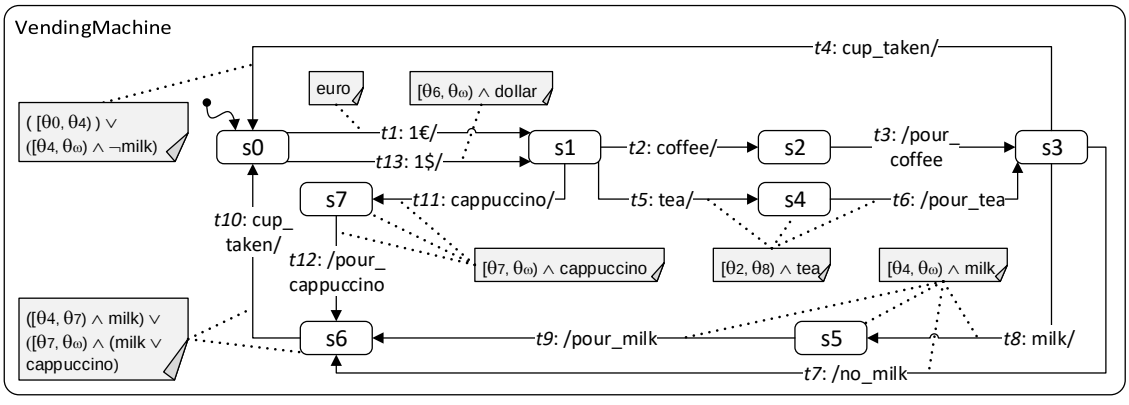


Abbildung 3.12.: Temporale Feature-annotierte State Machine $M_{175} = (M, \alpha_{175})$ eines Verkaufsautomaten

Im Folgenden wird das Vorgehen beim Slicing von 175%-Modellen anhand eines Beispiels demonstriert. Dazu ist in Abbildung 3.12 wieder die Feature-annotierte State Machine eines Verkaufsautomaten zu sehen, dessen Produktlinie sich über zehn Versionen erstreckt. Auf diese wird der Slicing-Algorithmus für ein Kriterium $C_{175} = (\varepsilon, \{milk \mapsto false, dollar \mapsto false\}, \theta_7)$ angewendet. Der Slice soll also eine State Machine für Version θ_7 ergeben und für eine Konfiguration, bei der weder das Feature `Milk` noch das Feature `Dollar` selektiert ist. Für e wird kein konkretes Element ausgewählt.

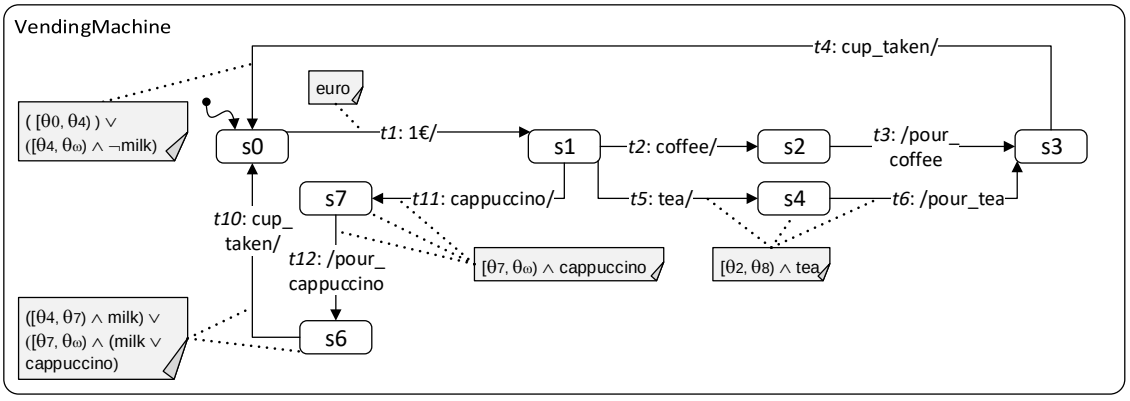


Abbildung 3.13.: Slice von M_{175} für $C_{175} = (\varepsilon, \{milk \mapsto false, dollar \mapsto false\}, \theta_7)$

Abbildung 3.13 zeigt den Slice von M_{175} für Kriterium C_{175}. Hierbei wurde durch eine fehlende Eingabe von e automatisch der Startzustand `s0` für den initialen Slice gewählt. Aufgrund der erstellten Abhängigkeiten wird Transition `t1` hinzugefügt, während Transition `t13` nicht hinzugefügt wird, da diese lediglich bei selektiertem Feature `Dollar` gültig ist. Um die Wohlgeformtheit zu erreichen wird Zustand `s1` hinzugefügt. In der nächsten Iteration werden anhand der Abhängigkeiten die Transitionen `t2`, `t5` und `t11` zum Slice hinzugefügt. Auch hier werden wieder die Zielzustände der Transitionen aufgrund der Wohlgeformtheit hinzugefügt. In ähnlicher Manier werden in der nächsten Iteration die Transitionen `t3`, `t6` und `t12` sowie die Zustände `s3` und `s6` hinzugefügt. Im nächsten Schritt wird Transition `t4` hinzugefügt, da sich für die Teilaussage $[\theta_4, \theta_\omega) \wedge \neg milk$ der Wahrheitswert *true* ergibt. Hingegen werden die Transitionen `t7` und `t8` nicht zum Slice hinzugefügt, da diese nur für Feature `Milk` gültig sind. Transition `t10` jedoch wird trotzdem hinzugefügt, da diese zwar bis Version θ_6 nur für Feature `Milk`, ab Version θ_7 jedoch auch für Feature `Cappuccino` gültig ist. Das bedeutet, auch wenn sich durch $\{milk \mapsto false\}$ und θ_7 als Kriterium für $[\theta_4, \theta_7) \wedge milk$ als Ergebnis $false$ ergibt, ist es doch immer noch möglich, dass sich für $[\theta_7, \theta_\omega) \wedge (milk \vee cappuccino)$ bei entsprechender Belegung von *cappuccino* als Ergebnis *true* ergibt. Aus diesem Grund muss auch `t10` im Slice enthalten sein. Da durch die fehlende Transition `t8` keine Abhängigkeiten mehr zu Zustand `s5` und Transition `t9` bestehen, gehören diese beiden Elemente nicht in den Slice. Somit ergibt sich ein Slice, der eine State Machine für θ_7 bildet. Da es sich bei der eingegebenen Konfiguration Γ lediglich um eine partielle Feature-Konfiguration handelt, ist das Ergebnis wiederum eine 150%-State Machine, da noch Entscheidungen bezüglich der Features `Tea` und `Cappuccino` getroffen werden müssen.

Wird das 175%-Slicing ohne explizite Definition eines Elementes im Kriterium für eine vollständige Konfiguration und genau eine Version durchgeführt, resultiert als Slice die Variante, die sich für die gewählte Konfiguration und Version ergibt. Besteht die Eingabe für das Kriterium aus einer partiellen Konfiguration oder einer Menge von Versionen, ergibt sich als Slice eine Menge von Varianten. Der Slice zeigt in diesem Fall also prinzipiell das Versionsfenster für die eingegebene Version, welches wiederum durch die gegebene (partielle) Konfiguration beschränkt wird. Durch die Angabe eines Elementes im Slicing-Kriterium kann die resultierende Variante oder Menge von Varianten weiterhin auf bestimmte Teile des Modells reduziert werden.

Nachdem in diesem Kapitel nun 175%-Modelle erläutert und formal definiert wurden, wird im nächsten Kapitel eine Möglichkeit untersucht, Higher-Order Delta-Modelle automatisch in 175%-Modelle umzuwandeln.

4 Transformation von Higher-Order Delta-Modellen in 175%-Modelle

Im letzten Kapitel wurde ein Ansatz konzipiert, Evolution von 150%-Modellen darzustellen, wodurch die 150%-Modelle auf 175%-Modelle erweitert wurden. In diesem Kapitel soll nun ein Algorithmus für die Transformation von Higher-Order Delta-Modellen in 175%-Modelle erarbeitet werden. Mit Hilfe eines solchen Algorithmus kann Zeit und Aufwand gespart werden, wenn etwa ein bereits bestehendes Higher-Order Delta-Modell zusätzlich als 175%-Modell repräsentiert werden soll oder wenn sowohl HOD-Modell als auch 175%-Modell parallel neu erstellt werden sollen. Dies kann der Fall sein, wenn der Bedarf besteht, eine Produktlinie aus unterschiedlichen Blickwinkeln zu betrachten oder aber verschiedene Analysen durchzuführen, die nicht auf der selben Art von Modell angewendet werden können.

In dieser Arbeit kann der Algorithmus angewendet werden, um Higher-Order Delta-Modelle von delta-basierten Evolutionsszenarien [29] in 175%-Modelle umzuwandeln. Diese 175%-Modelle können dann zur Evaluation der konzipierten Modellierungs- und Slicing-Methode verwendet werden. Des Weiteren kann diese Umwandlung zugleich zur Evaluation des Transformations-Algorithmus genutzt werden.

4.1. Definition von Transformationsregeln

Für die Konzeption eines Algorithmus zur Transformation von Higher-Order Delta-Modellen in 175%-Modelle wird zunächst allgemein erläutert und anhand von Beispielgrafiken verdeutlicht, wie sich die durch die Higher-Order Deltas vorgenommenen Änderungen in 175%-Modellen äußern. Dadurch werden Regeln für die Transformation der Higher-Order Delta-Modelle abgeleitet.

Dazu wird zunächst ein beispielhaftes Kernmodell gezeigt, welches im weiteren Verlauf immer wieder durch Deltas und Higher-Order Deltas verändert wird. Das Kernmodell ist in Abbildung 4.1 dargestellt und besteht aus vier Zuständen und fünf Transitionen. Bei diesen neun Elementen handelt es sich somit um Kernelemente, die für jede Feature-Konfiguration gültig sind. Aus diesem Grund erhalten die Kernelemente e für ihre Annotation $\alpha_{175}(e, \varphi(e))$ initial ein Tupel (θ, γ), wobei γ mit *true* belegt wird. Da das Kernmodell ab der ersten Version θ_0 existiert, welche zum Zeitpunkt dieser Betrachtung außerdem zunächst die einzige vorhandene Version darstellt, würde sich für das Tupel als θ daher θ_0 ergeben.

Kämen nun nacheinander weitere Versionen θ_1, θ_2 bis θ_i hinzu, dann würden für diese Versionen die Tupel $(\theta_1, true)$, $(\theta_2, true)$ bis $(\theta_i, true)$ zur Menge $\alpha_{175}(e, \varphi(e))$ hinzugefügt werden, sofern nicht durch ein Delta Änderungen an den Elementen e vorgenommen werden. Diese Tupel las-

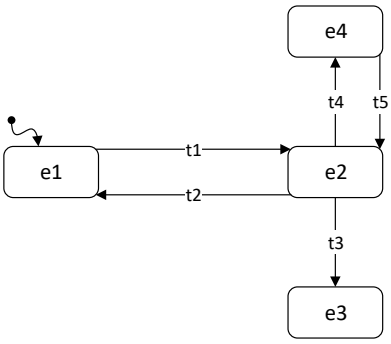


Abbildung 4.1.: Beispiel-Kernmodell

sen sich, wie in Kapitel 3.2 erläutert, zu einem Tupel $([\theta_0, \theta_i], true)$ zusammenfassen und somit als aussagenlogische Formel $[\theta_0, \theta_i] \wedge true$ darstellen. Zum Zeitpunkt des Hinzufügens der Elemente (Version θ_0) ist noch nicht bekannt, ob und wann die Elemente verändert werden, da sich diese Information erst aus der Anwendung von Deltas erschließt. Somit müsste für jede neue Version, welche die Elemente nicht verändert, ein neues Tupel mit dem neuen θ und dem alten γ hinzugefügt beziehungsweise das Intervall verändert werden. Ebenso verhält es sich für Elemente, die erst später durch Deltas hinzugefügt werden. Für eine Produktlinie, die insgesamt über hundert Elemente enthält, bei der aber beispielsweise nur fünf Elemente für eine neue Version verändert werden, würde es somit einen erheblichen Mehraufwand bedeuten, für jedes Element ein neues Tupel hinzuzufügen beziehungsweise das Intervall zu verändern.

Stattdessen besteht die Möglichkeit anstelle eines geschlossenen Intervalls ein rechtsoffenes Intervall zu verwenden und das Ende der Gültigkeit auf die hypothetische Version θ_ω (vgl. Kapitel 3.1) zu setzen. Dieses Intervall bleibt solange unangetastet bestehen, bis durch ein Delta eine Änderung an dem Element erfolgt. In dem Fall wird θ_ω ersetzt durch die Version, in der das besagte Delta zum ersten Mal auftaucht oder verändert wird. Demnach erhalten alle Kernelemente e bei der Transformation in ein 175%-Modell initial die Annotation

$$\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge true.$$

Diese Annotation kann im 175%-Modell wieder ausgeblendet werden, da sie insgesamt immer *true* ergibt. Abbildung 4.1 stellt somit gleichzeitig die initiale Form des 175%-Modells dar, welches dann durch Deltas und Higher-Order Deltas weiter mit Informationen angereichert wird.

Im Verlauf der Evolutionshistorie werden dabei durch Deltas weitere Elemente hinzugefügt, die nicht im Kernmodell enthalten sind. Diese Elemente besitzen zu Beginn keine Annotation, dargestellt durch

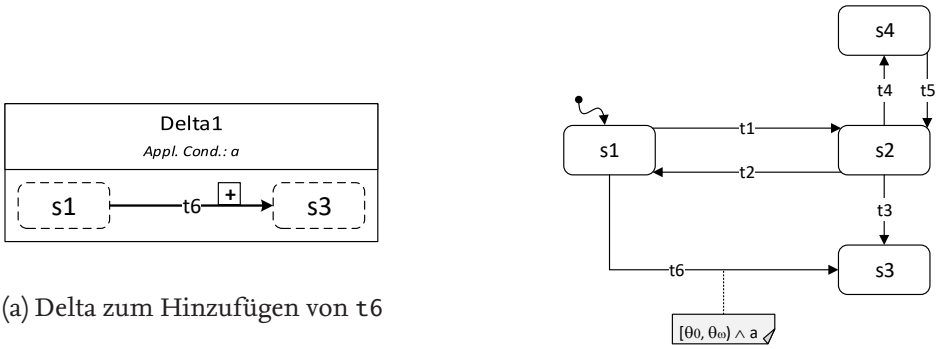
$$\alpha_{175}(e, \varphi(e)) = \bot .$$

Initiale Deltas

Zum Zeitpunkt der Version θ_0 existiert im Delta-Modell neben dem Kernmodell eine Menge von initialen Deltas. Diese Deltas können Elemente hinzufügen, entfernen oder modifizieren und werden unter einer bestimmten Anwendungsbedingung angewendet. Die Anwendungsbedingung definiert, für welche Features oder Kombinationen von Features das Delta verwendet werden darf.

Sie ist also die Voraussetzung für das Hinzufügen, Entfernen oder Modifizieren der Elemente und entscheidet somit über die Existenz oder Nichtexistenz der Elemente im Produktmodell. Daher ist die Anwendungsbedingung prädestiniert dafür, entweder direkt oder in negierter Form als γ für $\alpha_{175}(e, \varphi(e))$ verwendet zu werden.

Add-Operation



(a) Delta zum Hinzufügen von `t6`

(b) Annotation für `t6`

Abbildung 4.2.: Transformation eines Deltas mit hinzugefügtem Element

Abbildung 4.2a zeigt das Delta `Delta1`, welches die Transition `t6` zwischen den Zuständen `s1` und `s3` zum Kernmodell hinzufügt, wenn die Anwendungsbedingung a erfüllt ist. Da es sich bei diesem Delta um ein initiales Delta handelt und die aktuelle Version somit also immer noch θ_0 ist, ist die Transition `t6` ab θ_0 bis zu einer bis dato unbekannten Version θ_ω immer genau dann im Produktmodell enthalten, wenn Feature a selektiert ist. Dementsprechend wird `t6`, wie in Abbildung 4.2b dargestellt, mit der Annotation $\alpha_{175}(t6, \varphi(t6)) = [\theta_0, \theta_\omega) \wedge a$ zum 175%-Modell hinzugefügt. Allgemein ergibt sich somit für alle Elemente e, die durch ein initiales Delta hinzugefügt werden, die Annotation

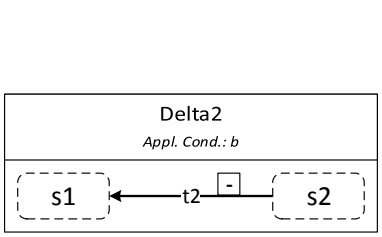
$$\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge \psi,$$

wobei ψ der Anwendungsbedingung des Deltas entspricht.

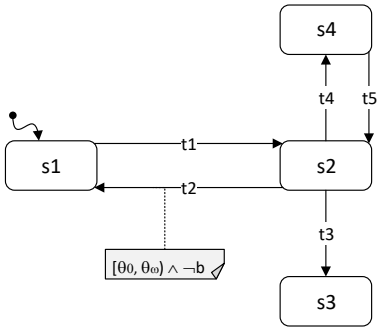
Remove-Operation

In Abbildung 4.3a ist das Delta `Delta2` dargestellt, welches die Transition `t2` zwischen den Zuständen `s1` und `s2` entfernt, wenn die Anwendungsbedingung b erfüllt ist. Bei `t2` handelt es sich um ein Element, welches bereits im Kernmodell vorhanden ist. `Delta2` ist erneut ein initiales Delta und die aktuelle Version ist immer noch θ_0. Aus diesen Umständen folgt, dass Transition `t2` ab θ_0 bis zu einer unbekannten Version θ_ω immer genau dann *nicht* im Produktmodell enthalten ist, wenn Feature b zu der Feature-Konfiguration des Produktmodells gehört. Umgekehrt bedeutet das, dass `t2` nur dann im Produktmodell auftaucht, wenn Feature b *nicht* selektiert ist. Daher wird die Annotation der Transition `t2` so verändert, dass das bisher existierende γ über eine Und-Verknüpfung mit der negierten Anwendungsbedingung ψ des Deltas verbunden wird, sodass

$$\gamma = \gamma \wedge \neg\psi.$$



(a) Delta zum Entfernen von t2



(b) Annotation für t2

Abbildung 4.3.: Transformation eines Deltas mit entferntem Element

Beispiel 2. *Transition* t2 *war zuvor mit* $\alpha_{175}(t2, \varphi(t2)) = [\theta_0, \theta_\omega) \wedge true$ *annotiert, da es sich um ein Kernelement handelt. Die vorherige Feature-Bedingung lautet somit* $\gamma = true$*. Die Anwendungsbedingung von* Delta2 *ist* $\psi = b$*. Entsprechend der Formel ergibt sich somit als neue Bedingung* $\gamma = true \wedge \neg b$*.*

Auf diese Weise erhält Transition t2 $\alpha_{175}(t2, \varphi(t2)) = [\theta_0, \theta_\omega) \wedge (true \wedge \neg b)$ als Annotation. Da der Wahrheitswert des neuen γ somit nur noch von $\neg b$ abhängt, kann das *true* weggelassen werden. So ergibt sich im 175%-Modell für t2 die Annotation $\alpha_{175}(t2, \varphi(t2)) = [\theta_0, \theta_\omega) \wedge \neg b$, dargestellt in Abbildung 4.3b.

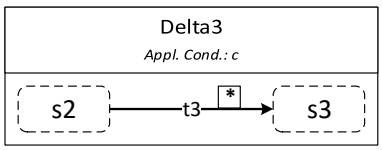
Theoretisch bestünde für diesen Fall also die Möglichkeit, *true* einfach durch $\neg b$ zu ersetzen. Da es sich bei dem zu entfernenden Element allerdings nicht zwangsläufig um ein im Kernmodell enthaltenes Element handeln muss, sondern auch ein Element entfernt werden kann, das zuvor durch ein anderes Delta hinzugefügt wurde, wird durch die Und-Verknüpfung sichergestellt, dass die vorherige Information nicht verloren geht. Wurde beispielsweise t6 aus Abbildung 4.2 unter der Bedingung a hinzugefügt, wird aber durch ein anderes Delta unter der Bedingung b wieder entfernt, ergibt sich als Annotation $[\theta_0, \theta_\omega) \wedge (a \wedge \neg b)$. Würde das alte γ ersetzt werden, ergebe sich als Annotation stattdessen $[\theta_0, \theta_\omega) \wedge \neg b$. Diese Annotation wäre für t6 allerdings inkorrekt, da t6 überhaupt nur im Produktmodell vorhanden sein kann, wenn zuallererst a gilt. Als weitere Einschränkung wird darüber hinaus dann $\neg b$ hinzugefügt, sodass gilt, dass t6 im Produktmodell enthalten ist, wenn a selektiert ist, aber nicht b. Als allgemeine Formel für die Annotation von Elementen e, die durch ein initiales Deltas entfernt werden, ergibt sich somit

$$\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge (\gamma \wedge \neg\psi),$$

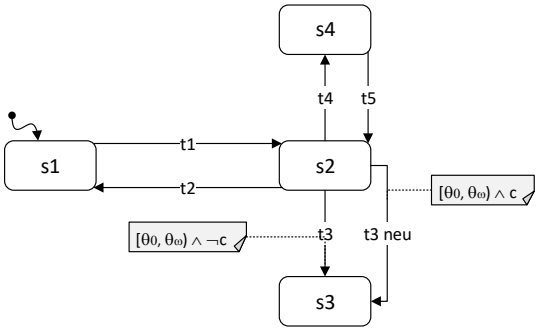
wobei γ an dieser Stelle der vorherigen Bedingung entspricht.

Modify-Operation

Abbildung 4.4a zeigt das Delta Delta3, welches die Transition t3 zwischen den Zuständen s2 und s3 modifiziert, wenn das Feature c selektiert ist. Transition t3 ist wieder ein Element des Kernmodells und Delta3 ist wieder ein initiales Delta. Daraus ergibt sich, dass t3 ab θ_0 bis zu einer unbekannten Version θ_ω immer dann modifiziert wird, wenn Feature c selektiert ist. Das bedeutet, dass Transition t3 in ihrer originalen Form genau dann im Produktmodell enthalten ist, wenn Bedingung c



(a) Delta zum Modifizieren von `t3`



(b) Annotation für `t3`

Abbildung 4.4.: Transformation eines Deltas mit modifiziertem Element

nicht zutrifft, während die modifizierte Ausführung der Transition im Produktmodell enthalten ist, wenn Bedingung c zutrifft. Im 175%-Modell werden die verschiedenen Versionen von `t3` als eigene Transitionen repräsentiert. Die originale Transition wird dabei behandelt wie eine Transition, die entfernt wird. Daher ergibt sich für `t3` die Annotation $\alpha_{175}(t3, \varphi(t3)) = [\theta_0, \theta_\omega) \wedge (true \wedge \neg c)$. Diese ist im 175%-Modell in Abbildung 4.4b in gekürzter Form $[\theta_0, \theta_\omega) \wedge \neg c$ dargestellt. Für die modifizierte Ausführung `t3neu` lautet die Annotation $\alpha_{175}(t3neu, \varphi(t3neu)) = [\theta_0, \theta_\omega) \wedge c$. Im Allgemeinen setzt sich die Annotation für das zu modifizierende Element e_{bm} (*before modification*) also zusammen als

$$\alpha_{175}(e_{bm}, \varphi(e_{bm})) = [\theta_0, \theta_\omega) \wedge (\gamma \wedge \neg\psi),$$

während sich für das modifizierte Element e_{am} (*after modification*) die Annotation

$$\alpha_{175}(e_{am}, \varphi(e_{am})) = [\theta_0, \theta_\omega) \wedge (\gamma_{e_{bm}} \wedge \psi)$$

ergibt. $\gamma_{e_{bm}}$ bezeichnet hier die Bedingung des originalen Elements e_{bm} vor der Modifikation. Auf diese Weise wird sichergestellt, dass für e_{am} auch grundsätzlich erst einmal die Bedingungen erfüllt werden, durch die das originale Element e_{bm} überhaupt erst im Modell vorhanden ist.

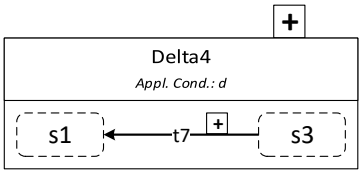
Nachdem nun Regeln für die Transformation von initial im Delta-Modell enthaltenen Deltas aufgestellt wurden, werden im Folgenden Deltas betrachtet, die durch Higher-Order Deltas zum Delta-Modell hinzugefügt werden.

Hinzugefügte Deltas

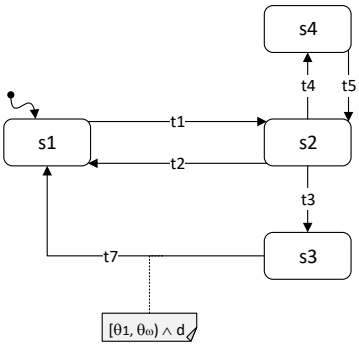
Für die Betrachtung wird angenommen, dass die Produktlinie um die Version θ_1 erweitert wurde. Die folgenden Deltas werden alle in Version θ_1 dem Delta-Modell hinzugefügt. Auch diese Deltas können wiederum Elemente hinzufügen, entfernen oder modifizieren.

Add-Operation

Das hinzugefügte Delta `Delta4` in Abbildung 4.5a fügt die Transition `t7` zwischen den Zuständen `s3` und `s1` zum Kernmodell hinzu, wenn die Anwendungsbedingung d erfüllt ist. Da dieses Delta in Version θ_1 zum Delta-Modell hinzugefügt wurde, ist die Transition `t7` ab θ_1 bis zu einer bis dato unbekannten Version θ_ω immer genau dann im Produktmodell enthalten, wenn Feature d



(a) Delta zum Hinzufügen von `t7`



(b) Annotation für `t7`

Abbildung 4.5.: Transformation eines hinzugefügten Deltas mit hinzugefügtem Element

selektiert ist. Somit wird also die Transition `t7` mit der Annotation $\alpha_{175}(t7, \varphi(t7)) = [\theta_1, \theta_\omega) \wedge d$ zum 175%-Modell - dargestellt in Abbildung 4.5b - hinzugefügt.

Transition `t7` wird durch `Delta4` zum ersten Mal in der Historie hinzugefügt und war somit zuvor noch nicht im 175%-Modell enthalten. Gerade im späteren Verlauf einer Historie kann es allerdings passieren, dass durch ein hinzugefügtes Delta ein Element hinzugefügt wird, welches bereits im 175%-Modell enthalten ist, da es für ein früheres Intervall schon einmal gültig war. In diesem Fall darf die Annotation nicht einfach ersetzt werden, da sonst die Informationen für die vorherige Gültigkeit verloren gehen würden. Aus diesem Grund muss die neue Kombination aus Intervall und Anwendungsbedingung als neues Tupel zur alten Annotation hinzugefügt werden. Das bedeutet, die alte Annotation muss mit der neuen Annotation über eine Oder-Verknüpfung verbunden werden.

Beispiel 3. *Das Delta `Delta4` wird erst für Version θ_6 hinzugefügt, aber `t7` ist bereits mit der Annotation $\alpha_{175}(t7, \varphi(t7)) = [\theta_1, \theta_4) \wedge a$ im 175%-Modell enthalten. Für `t7` ergibt sich somit $\alpha_{175}(t7, \varphi(t7)) = ([\theta_1, \theta_4) \wedge a) \vee ([\theta_6, \theta_\omega) \wedge d)$ als Annotation im 175%-Modell.*

An dieser Stelle wird davon ausgegangen, dass das letzte Intervall der alten Annotation bereits vorher abgeschlossen wurde. Sollte das nicht der Fall sein, würde das bedeuten, dass dasselbe Element zur gleichen Zeit noch durch ein anderes Delta - vermutlich unter einer anderen Anwendungsbedingung - in das Kernmodell eingefügt werden kann. Dies kann dazu führen, dass für bestimmte Konfigurationen versucht wird, dasselbe Element durch zwei unterschiedliche Deltas zum Kernmodell hinzuzufügen. Dieser Vorgang beschreibt einen Konflikt, weshalb das Hinzufügen eines Elementes durch mehrere unterschiedliche Deltas hier als nicht zulässig eingestuft wird. Das bedeutet, in jeder Version der Softwareproduktlinie darf für jedes Element höchstens ein Delta existieren, welches das Element hinzufügt.

Daraus folgt, dass ein Delta, welches eine Add-Operation auf e ausführt, spätestens in der Version entfernt werden muss, in der ein neues Delta hinzugefügt wird, welches ebenfalls eine Add-Operation auf e ausführt. Wird das zuerst existierende Delta dabei in einer Version θ_h entfernt und das neue Delta in θ_i hinzugefügt, wobei gilt, dass $\theta_h < \theta_i$, dann ist sichergestellt, dass das letzte Intervall der Annotation von e bereits abgeschlossen ist, wenn das neue Delta hinzugefügt wird. Wird nämlich ein Delta, welches eine Add-Operation auf e ausführt, aus dem Delta-Modell entfernt, ohne

dass ein anderes Delta, welches ebenfalls eine Add-Operation auf e ausführt, in der selben Version hinzugefügt wird, müssen konsequent alle weiteren Deltas entfernt werden, die Operationen auf e ausführen. Anderenfalls könnten Remove- oder Modify-Operationen auf ein nicht im Modell existierendes Element angewendet werden, was einen Konflikt darstellt. Daher ist davon auszugehen, dass keine weiteren Einschränkungen für ein durch ein hinzugefügtes Delta hinzugefügtes Element e bestehen, wenn das letzte Intervall der Annotation bereits abgeschlossen wurde. In diesem Fall muss daher keine vorherige Feature-Bedingung in Betracht bezogen werden. Somit können das neue Intervall und die Anwendungsbedingung des Deltas einfach über eine Oder-Verknüpfung an die bisherige Annotation angehängt werden.

Andererseits besteht die Möglichkeit, dass das zuerst existierende Delta, welches Element e hinzufügt, erst in derselben Version aus dem Delta-Modell entfernt wird, in der das neue Delta hinzugefügt wird. In diesem Fall kann es sein, dass noch weitere Deltas im Delta-Modell existieren, welche die Existenz von e im Modell einschränken, das heißt, unter bestimmten Bedingungen e entfernen oder modifizieren. Wird im selben Schritt ein Delta entfernt, welches e hinzufügt, und ein Delta hinzugefügt, welches e ebenfalls hinzufügt, dann können die Deltas, welche e modifizieren oder entfernen, im Delta-Modell enthalten bleiben. In diesem Fall ist nämlich das Element, auf welches sie sich beziehen, ohne Unterbrechung im Delta-Modell existent.

Beispiel 4. *In Version θ_0 existiert `DeltaX`, welches Element e mit der Bedingung $\psi = x$ hinzufügt. Des Weiteren existieren `DeltaY` und `DeltaZ`, welche e mit den Bedingungen $\psi = y$ und $\psi = z$ entfernen beziehungweise modifizieren. Nach den Regeln für initiale Deltas ergibt sich für e die Annotation $\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge (x \wedge \neg y \wedge \neg z)$. In Version θ_1 wird `DeltaX` entfernt. Folglich müssten auch `DeltaY` und `DeltaZ` entfernt werden, da sie auf e operieren, e jedoch durch kein Delta mehr in das Produktmodell eingefügt wird. Stattdessen wird `DeltaQ` hinzugefügt, welches e mit der Bedingung $\psi = q$ hinzufügt. Die Voraussetzung für `DeltaY` und `DeltaZ` ist somit wieder gegeben, wodurch diese im Delta-Modell verbleiben dürfen.*

Dementsprechend können die Einschränkungen durch diese Deltas aus der Bedingung der alten Annotation von e übernommen werden. Genauer gesagt, müssen die Einschränkungen sogar übernommen werden, da es sich sonst um eine fehlerhafte Umwandlung des Delta-Modells handelt, wenn die Einschränkungen auf e durch die Deltas noch bestehen, aber nicht in der Annotation umgesetzt werden. Da bereits umgewandelte Deltas allerdings nicht noch einmal erneut betrachtet werden, sofern keine Operation auf ihnen erfolgt, können die Einschränkungen nur aus der alten Annotation des Elements übernommen werden. Die neue Feature-Bedingung wird für diesen Fall daher gebildet, indem die Anwendungsbedingung des Deltas über eine Und-Verknüpfung an die alte Feature-Bedingung angehängt wird. Auf diese Weise werden sowohl die bisher bestehenden Einschränkungen als auch die neue Operation berücksichtigt.

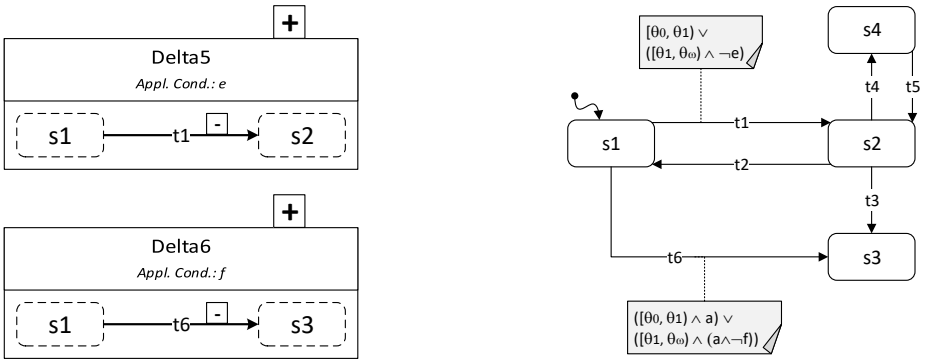
Für die Annotation von Elementen e, welche durch ein hinzugefügtes Delta zum Produktmodell hinzugefügt werden, werden folglich einige Fallunterscheidungen getroffen:

$$\alpha_{175}(e, \varphi(e)) = \begin{cases} [\theta_i, \theta_\omega) \wedge \psi & \text{wenn } \alpha_{175}(e, \varphi(e)) = \bot \\ \alpha_{175}(e, \varphi(e)) \vee ([\theta_i, \theta_\omega) \wedge \psi) & \text{wenn } \alpha_{175}(e, \varphi(e)) \neq \bot \wedge \theta_u \leq \theta_i \\ \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \psi)) & \text{wenn } \alpha_{175}(e, \varphi(e)) \neq \bot \wedge \theta_u = \theta_\omega \end{cases}$$

Neben der Unterscheidung, ob $\theta_u \leq \theta_i$ oder $\theta_u = \theta_\omega$, wird durch $\alpha_{175}(e, \varphi(e)) = \bot$ beziehungsweise $\alpha_{175}(e, \varphi(e)) \neq \bot$ außerdem differenziert, ob dem Element vorher schon eine Annotation zu-

geordnet war oder nicht. Sollte zuvor noch keine Annotation zugeordnet gewesen sein, weil das Element zum ersten Mal in der Historie hinzugefügt wird, oder bei einer bestehenden Annotation mit ausschließlich abgeschlossenen Intervallen, muss natürlich auch keine vorherige Bedingung beachtet werden.

Remove-Operation



(a) Deltas zum Entfernen von `t1` und `t6`

(b) Annotationen für `t1` und `t6`

Abbildung 4.6.: Transformation eines hinzugefügten Deltas mit entferntem Element

Abbildung 4.6a zeigt die zwei hinzugefügten Deltas `Delta5` und `Delta6`, die je eine Transition entfernen. `Delta5` entfernt die bereits im Kernmodell enthaltene Transition `t1`, wenn Feature e selektiert ist, während `Delta6` die durch `Delta1` hinzugefügte Transition `t6` entfernt, wenn Feature f selektiert ist. Auch diese beiden Deltas werden in Version θ_1 zum Delta-Modell hinzugefügt. Somit ist die Transition `t1` ab θ_1 bis zu einer bis dato unbekannten Version θ_ω immer dann *nicht* im Produktmodell enthalten, wenn die Bedingung e erfüllt ist. Das heißt, die Transition ist ab θ_1 nur dann im Produktmodell enthalten, wenn e *nicht* zutrifft. Selbiges gilt für Transition `t6` unter Bedingung f.

Für beide Transitionen ändert sich γ daher ab Einfügen des Deltas, um die neue Bedingung mit berücksichtigen zu können. Aus diesem Grund wird das zuvor bestehende Intervall, auf das sich das alte γ bezieht, abgeschlossen. Das heißt, das Ende des Intervalls wird auf die Version gesetzt, in der das Delta eingefügt wird (in diesem Fall θ_1). Des Weiteren wird eine Oder-Verknüpfung erstellt mit einem Intervall, dass ab genau dieser Version beginnt und bis zu einer unbekannten Version θ_ω gültig ist. Als neues γ ergibt sich wieder

$$\gamma = \gamma \wedge \neg\psi.$$

Das alte γ wird also über eine Und-Verknüpfung mit der negierten Anwendungsbedingung des Deltas verbunden.

Beispiel 5. *Die alte Annotation von Transition `t1` lautet $\alpha_{175}(t1, \varphi(t1)) = [\theta_0, \theta_1)$. Die alte Feature-Bedingung ist somit $\gamma = true$, welche lediglich ausgeblendet wurde. Die Anwendungsbedingung von `Delta5` ist $\psi = e$. Als neue Feature-Bedingung resultiert somit $\gamma = true \wedge \neg e$, beziehungsweise $\gamma = \neg e$, da true wieder ausgeblendet werden kann. Analog ergibt sich für Transition `t6`, welche zuvor die Feature-Bedingung $\gamma = a$ besaß, die neue Feature-Bedingung $\gamma = a \wedge \neg f$.*

Für Transition `t1` ergibt sich somit $\alpha_{175}(t1, \varphi(t1)) = [\theta_0, \theta_1) \vee ([\theta_1, \theta_\omega) \wedge \neg e)$ als neue Annotation, während `t6` die Annotation $\alpha_{175}(t6, \varphi(t6)) = ([\theta_0, \theta_1) \wedge a) \vee ([\theta_1, \theta_\omega) \wedge (a \wedge \neg f))$ erhält. Beide Transitionen und ihre neuen Annotationen sind in Abbildung 4.6b dargestellt.

Im Allgemeinen ergibt sich für Elemente e, die durch ein hinzugefügtes Delta entfernt werden, somit die Formel

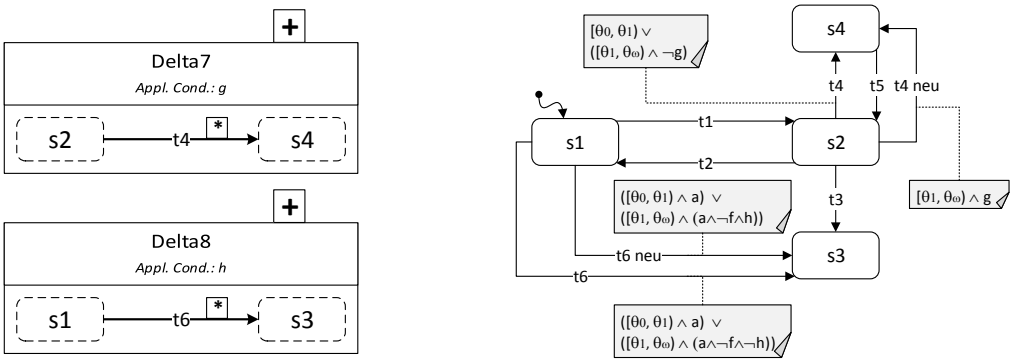
$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \neg\psi))$$

als Annotation. $\alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ beschreibt dabei die angepasste Form der alten Annotation, bei der θ_u des letzten Intervalls auf θ_i gesetzt wurde. Dies ist hier immer der Fall, da e auf jeden Fall schon mit einer Annotation im 175%-Modell existieren muss. Anderenfalls könnte das Element nicht durch eine Operation entfernt werden, wenn es nicht entweder im Kernmodell enthalten ist oder durch ein anderes Delta hinzugefügt wurde. In beiden Fällen hat das Element bereits eine Annotation erhalten. Des Weiteren enthält diese Annotation das letzte Intervall immer in nicht abgeschlossener Form, also von einer beliebigen Version $\theta_{\leq i}$ bis θ_ω. Dies liegt daran, dass eine Remove-Operation auf einem Element nur dann angewendet werden kann, wenn das Element zuvor durch ein anderes Delta zum Modell hinzugefügt wurde. Dieses andere Delta muss zeitgleich mit dem neu hinzugefügten Delta gültig sein. Wäre das letzte Intervall in der Annotation des Elements nun bereits vorher abgeschlossen wurden, können das Delta zum Hinzufügen des Elements und das Delta zum Entfernen des Element auf keinen Fall gleichzeitig gültig sein. Daraus ergibt sich, dass bei einem durch ein hinzugefügtes Delta entferntes Element e das letzte Intervall der bisherigen Annotation immer in nicht abgeschlossener Form vorliegen muss. Ist dies nicht der Fall, ist bereits das Delta-Modell nicht konfliktfrei.

Das letzte Intervall darf dabei ab einer Version $\theta_{\leq i}$, also einer Version, die entweder θ_i oder älter als θ_i ist, beginnen. Dadurch wird ermöglicht, dass in einer Version mehrere Deltas eingefügt werden können, die einschränkende Operationen (*Add* oder *Modify*) auf einem Element e ausführen. Somit entstehen in einer Version mehrere Intervalle der Form $[\theta_i, \theta_i)$, wodurch die zugeordneten Feature-Bedingungen keine Gültigkeit besitzen, da die Intervalle durch $[\theta_i, \theta_i) = \{\ \}$ keine gültigen Versionen enthalten. Die Feature-Bedingung für das jeweils nächste Intervall basiert trotzdem auf der letzten Feature-Bedingung, auch wenn sie solch einem ungültigen Intervall zugeordnet ist. Die letzte Änderung einer Annotationfür die Version θ_i enthält dementsprechend alle relevanten Informationen über die Delta-Änderungen in θ_i, die Auswirkungen auf e haben. Die ungültigen Intervalle mit zugeordneter Feature-Bedingung können nach Abschluss der Version aus der Annotation entfernt werden.

Beispiel 6. *Es existiert ein Element e mit der Annotation $\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge z$. In Version θ_i werden zwei Deltas `DeltaP` und `DeltaQ` zum Delta-Modell hinzugefügt, die e einmal für Bedingung p und einmal für Bedingung q aus dem Modell entfernen. Bei Betrachtung von `DeltaP` wird die Annotation von e entsprechend der oben definierten Formel zunächst in $\alpha_{175}(e, \varphi(e)) = ([\theta_0, \theta_i) \wedge z) \vee ([\theta_i, \theta_\omega) \wedge (z \wedge \neg p))$ geändert. Bei der darauffolgenden Betrachtung von `DeltaQ` wird die Annotation wiederum in $\alpha_{175}(e, \varphi(e)) = ([\theta_0, \theta_i) \wedge z) \vee ([\theta_i, \theta_i) \wedge (z \wedge \neg p)) \vee ([\theta_i, \theta_\omega) \wedge (z \wedge \neg p \wedge \neg q))$ geändert. Hierbei verliert der mittlere Teil $[\theta_i, \theta_i) \wedge (z \wedge \neg p)$ aufgrund des Intervalls $[\theta_i, \theta_i)$ jegliche Gültigkeit. Der mittlere Teil kann daher aus der Annotation entfernt werden, sodass sich $\alpha_{175}(e, \varphi(e)) = ([\theta_0, \theta_i) \wedge z) \vee ([\theta_i, \theta_\omega) \wedge (z \wedge \neg p \wedge \neg q))$ als Annotation für e ergibt. Der letzte Teil $[\theta_i, \theta_\omega) \wedge (z \wedge \neg p \wedge \neg q)$ enthält alle relevanten Informationen über die Änderungen auf e in θ_i.*

Modify-Operation



(a) Deltas zum Modifizieren von `t4` und `t6` (b) Annotationen für `t4` und `t6`

Abbildung 4.7.: Transformation eines hinzugefügten Deltas mit modifiziertem Element

In Abbildung 4.7a sind die zwei hinzugefügten Deltas `Delta7` und `Delta8` dargestellt, welche die Transitionen `t4` und `t6` modifizieren. Transition `t4` ist bereits im Kernmodell enthalten und wird modifiziert, wenn Bedingung g erfüllt ist. Hingegen wurde `t6` in Version θ_0 erst durch `Delta1` hinzugefügt, wird seit θ_1 durch `Delta6` entfernt und wird nun modifiziert, wenn Bedingung h erfüllt ist. `Delta7` und `Delta8` werden beide in Version θ_1 zum Delta-Modell hinzugefügt. Daraus folgt, dass Transition `t4` ab θ_1 bis zu einer bis dato unbekannten Version θ_ω immer dann modifiziert wird, wenn Feature g selektiert ist. Somit ist `t4` in ihrer originalen Form genau dann im Produktmodell enthalten, wenn Bedingung g *nicht* zutrifft, während die modifizierte Ausführung der Transition im Produktmodell enthalten ist, wenn Bedingung g zutrifft. Das gleiche Prinzip gilt für Transition `t6` unter Bedingung h. Die verschiedenen Versionen von `t4` und `t6` sind im 175%-Modell als unterschiedliche Transitionen repräsentiert. Mit originalen Transitionen wird dabei umgegangen wie mit Transitionen, die durch ein hinzugefügtes Deltas entfernt werden.

Dementsprechend ergibt sich für Transition `t4` die Annotation $[\theta_0, \theta_1) \vee ([\theta_1, \theta_\omega) \wedge \neg g)$ im 175%-Modell und für die modifizierte Version `t4neu` ergibt sich die Annotation $[\theta_1, \theta_\omega) \wedge g$. Auch für Transition `t6` wird in der zuvor existierenden Annotation $([\theta_0, \theta_1) \wedge a) \vee ([\theta_1, \theta_\omega) \wedge (a \wedge \neg f))$ das Intervall abgeschlossen und diese alte Annotation über eine Oder-Verknüpfung mit der neuen Formel $[\theta_1, \theta_\omega) \wedge (a \wedge \neg f \wedge \neg h)$ verbunden. Dadurch ergibt sich für `t6` im 175%-Modell $([\theta_0, \theta_1) \wedge a) \vee ([\theta_1, \theta_1) \wedge (a \wedge \neg f)) \vee ([\theta_1, \theta_\omega) \wedge (a \wedge \neg f \wedge \neg h))$ als Annotation. Da die Transition in Version θ_1 bereits neu durch ein Delta entfernt und in der selben Version direkt wieder durch ein neues Delta modifiziert wurde, und das erste Intervall somit seine Gültigkeit gänzlich verloren hat, wird das Intervall samt des verknüpften γ komplett entfernt. Folglich resultiert für `t6` die gekürzte Formel $([\theta_0, \theta_1) \wedge a) \vee ([\theta_1, \theta_\omega) \wedge (a \wedge \neg f \wedge \neg h))$ als Annotation. Die Transitionen `t4` und `t7` samt Annotationen sind in Abbildung 4.7b dargestellt. Als allgemeine Formel für das zu modifizierende Element e_{bm} ergibt sich

$$\alpha_{175}(e_{bm}, \varphi(e_{bm})) = \alpha_{175}(e_{bm}, \varphi(e_{bm}))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \neg\psi))$$

für die Annotation. Auch an dieser Stelle gilt, dass das Element bereits eine Annotation mit einem nicht abgeschlossenen, letzten Intervall besitzen muss, um überhaupt modifiziert werden zu

dürfen. Die allgemeine Annotation für das modifizierte Element e_{am} lautet

$$\alpha_{175}(e_{am}, \varphi(e_{am})) = \begin{cases} [\theta_i, \theta_\omega) \wedge (\gamma_{e_{bm}} \wedge \psi) & \text{wenn } \alpha_{175}(e_{am}, \varphi(e_{am})) = \bot \\ \alpha_{175}(e_{am}, \varphi(e_{am})) \vee ([\theta_i, \theta_\omega) \wedge (\gamma_{e_{bm}} \wedge \psi)) & \text{sonst.} \end{cases}$$

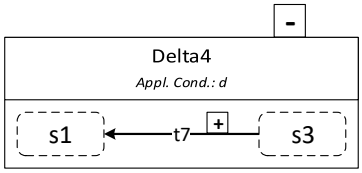
Durch die Fallunterscheidung wird zusätzlich berücksichtigt, dass das modifizierte Element auch zu einem früheren Zeitpunkt schon einmal gültig gewesen sein kann, aber nicht muss.

Nach der Formulierung der Regeln für die Transformation von zum Delta-Modell hinzugefügten Deltas, werden im nächsten Abschnitt Deltas untersucht, die durch Higher-Order Deltas aus dem Delta-Modell entfernt werden.

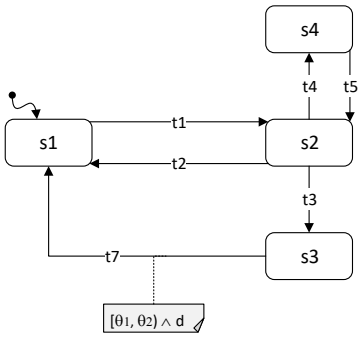
Entfernte Deltas

Für die Untersuchung der entfernten Deltas wird angenommen, dass die Version θ_2 zur Produktlinie hinzugefügt wurde. Die Deltas in diesem Abschnitt werden alle in Version θ_2 aus dem Delta-Modell entfernt. Die entfernten Deltas haben zuvor wiederum Elemente hinzugefügt, entfernt oder modifiziert.

Add-Operation



(a) Delta zum Hinzufügen von `t7`



(b) Annotation für `t7`

Abbildung 4.8.: Transformation eines entfernten Deltas mit hinzugefügtem Element

Abbildung 4.8a zeigt `Delta4`, welches in Version θ_2 wieder entfernt wird, nachdem es in θ_1 hinzugefügt wurde. Das bedeutet, dass `Delta4` seit dem Hinzufügen des Deltas die Transition `t7` solange in das Produktmodell einfügt, bis das Delta wieder entfernt wird. Da `t7` nicht ersatzweise durch ein anderes Delta hinzugefügt wird, wird nach dem Entfernen von `Delta4` Transition `t7` nicht mehr zu einem Produktmodell hinzugefügt und ist folglich in keinem Produktmodell von Version θ_2 enthalten. Daher wird für `t7` das letzte Intervall der bisherigen Annotation abgeschlossen und kein neues Intervall begonnen. Für `t7` ergibt sich daher $[\theta_1, \theta_2) \wedge d$ als Annotation im 175%-Modell, dargestellt in Abbildung 4.8b.

Angenommen in Version θ_2 wurde bereits als Ersatz ein anderes Deltas eingefügt, das `t7` unter einer anderen Bedingung zum Produktmodell hinzufügt. Diese Bedingung wurde als neues γ^+, also als Bedingung, durch die `t7` zum Modell hinzugefügt wird, gespeichert. Gleichzeitig hat sich durch die Regeln für hinzugefügte Deltas die Feature-Bedingung $\gamma = \psi \wedge \gamma^+$ für `t7` ergeben, wobei

ψ die Anwendungsbedingung von `Delta4` bezeichnet. Wird nun `Delta4` entfernt, existiert Transition `t7` in Zukunft dennoch in einigen Produktmodellen, da sie durch das Ersatzdelta hinzugefügt wird. Die Annotation von `t7` darf somit nicht wie im anderen Fall komplett abgeschlossen werden. Stattdessen muss wieder ein neues Intervall in Kombination mit einer neuen Feature-Bedingung an die Annotation angehängt werden. Als neue Feature-Bedingung ergibt sich in diesem Fall

$$\gamma = \gamma^{-\psi}.$$

Dies ist die alte Feature-Bedingung von `t7`, aus der die Anwendungsbedingung ψ von `Delta4` entfernt wurde.

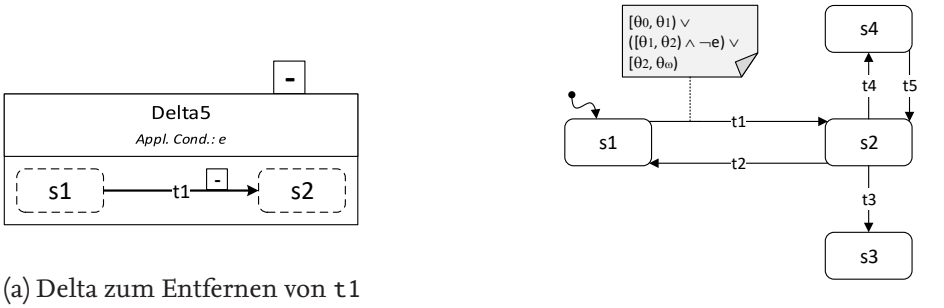
Beispiel 7. *Transition `t7` besitzt eine Annotation $[\theta_1, \theta_\omega) \wedge d \wedge s$. Die alte Feature-Bedingung von `t7` lautet folglich $\gamma = d \wedge s$, wobei $\gamma^+ = s$. Da $\psi = d$ die Anwendungsbedingung von `Delta4` ist, ergibt sich als $\gamma^{-\psi} = s$. Die neue Feature-Bedingung von `t7` lautet somit $\gamma = s$. Als neue Annotation von `t7` resultiert somit $([\theta_1, \theta_2) \wedge (d \wedge s)) \vee ([\theta_2, \theta_\omega) \wedge s)$.*

Für Add-Operationen in entfernten Deltas erfolgt somit eine Fallunterscheidung zwischen $\gamma^+ \neq \psi$, wenn bereits ein neueres Delta das Hinzufügen des betroffenen Elements e übernimmt, oder $\gamma^+ = \psi$, wenn das entfernte Delta das einzige Delta ist, dass eine Add-Operation auf dem Element ausführt. Im Allgemeinen ergibt sich daher

$$\alpha_{175}(e, \varphi(e)) = \begin{cases} \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge \gamma^{-\psi}) & \text{wenn } \gamma^+ \neq \psi \\ \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} & \text{sonst} \end{cases}$$

als Annotation für durch entfernte Deltas hinzugefügte Elemente e.

Remove-Operation



(a) Delta zum Entfernen von `t1`

(b) Annotation für `t1`

Abbildung 4.9.: Transformation eines entfernten Deltas mit entferntem Element

Das in Version θ_1 hinzugefügte `Delta5` wird in Abbildung 4.9a wieder entfernt. Die Transition `t1`, die durch `Delta5` entfernt wird, wenn Bedingung e erfüllt ist, wird somit nach Entfernen des Deltas nicht mehr entfernt, sofern sie nicht wiederum durch ein anderes Delta entfernt wird. Das bedeutet, dass die Transition in Zukunft auch wieder unter der Bedingung e im Produktmodell existiert. Daraus ergibt sich, dass das letzte Intervall der bisherigen Annotation abgeschlossen wird und über eine Oder-Verknüpfung eine neue Formel an die alte Annotation angehängt wird. Die

neue Formel besteht aus einem Intervall, das ab der Version, in der das Delta entfernt wird, bis zu einer unbekannten Version θ_ω gültig ist, sowie aus der Bedingung $\gamma^{-(\neg\psi)}$. Diese Bedingung ist die vorherige Bedingung γ, aus der jedoch die negierte Anwendungsbedingung des entfernten Deltas wieder entfernt wurde.

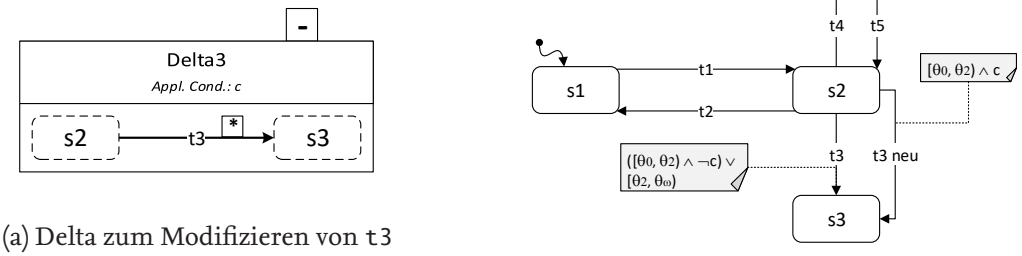
Beispiel 8. *Für Transition* `t1` *lautet die alte Annotation* $[\theta_0, \theta_1) \vee ([\theta_1, \theta_2) \wedge \neg e)$*. Die vorherige Feature-Bedingung ist somit* $\gamma = \neg e$*, oder genauer* $\gamma = true \wedge \neg e$*. Die Anwendungsbedingung von* `Delta5` *ist* $\psi = e$*. Das Entfernen der negierten Anwendungsbedingung aus* γ *ergibt somit an dieser Stelle* $\gamma^{-(\neg\psi)} = true$*.*

Auf diese Weise erhält Transition `t1` die neue Annotation $[\theta_0, \theta_1) \vee ([\theta_1, \theta_2) \wedge \neg e) \vee [\theta_2, \theta_\omega)$, wie in Abbildung 4.9b dargestellt. Folglich bildet sich

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge \gamma^{-(\neg\psi)})$$

als allgemeine Form für die Annotation jener Elemente e, die durch ein entferntes Delta entfernt werden.

Modify-Operation



(a) Delta zum Modifizieren von `t3`

(b) Annotation für `t3`

Abbildung 4.10.: Transformation eines entfernten Deltas mit modifiziertem Element

In Abbildung 4.10a wird das initiale Delta `Delta3` entfernt. Nach Entfernen des Deltas wird die Transition `t3`, die durch das Delta modifiziert wird, wenn Feature c selektiert ist, daher nicht mehr modifiziert, sofern sie nicht durch ein anderes Delta modifiziert wird. Die sich durch `Delta3` ergebende, modifizierte Version der Transition ist folglich nicht mehr gültig ab der Version, in der das Delta entfernt wird (in diesem Fall θ_2). Die originale Transition `t3` wiederum, ist ab dieser Version auch wieder im Produktmodell enthalten, wenn Bedingung c erfüllt ist. Dementsprechend wird bei der modifizierten Transition das letzte Intervall mit θ_2 abgeschlossen und sonst keine weitere Änderung an der Annotation vorgenommen. Die modifizierte Version der Transition wird somit behandelt wie eine Transition, die durch ein entferntes Delta hinzugefügt wird. Für die zu modifizierende Version der Transition wird ebenfalls das letzte Intervall abgeschlossen. Jedoch wird hierbei zusätzlich ein neues Intervall von θ_2 bis zu einer unbekannten Version θ_ω zusammen mit der Bedingung $\gamma^{-(\neg\psi)}$ über eine Oder-Verknüpfung mit der bisherigen Annotation verbunden. Die zu modifizierende Transition wird folglich wie eine Transition gehandhabt, die durch ein entferntes Delta entfernt wird.

Die modifizierte Transition t3neu erhält im 175%-Modell in Abbildung 4.10b dementsprechend die Annotation $[\theta_0, \theta_2) \wedge c$, während sich für die originale Transition t3 die Annotation $([\theta_0, \theta_2) \wedge \neg c) \vee [\theta_2, \theta_\omega)$ ergibt. Im Allgemeinen resultiert somit

$$\alpha_{175}(e_{bm}, \varphi(e_{bm})) = \alpha_{175}(e_{bm}, \varphi(e_{bm}))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge \gamma^{-(\neg\psi)})$$

als Annotation für das originale Element e_{bm}. Für das modifizierte Element e_{am} lautet die allgemeine Form der Annotation

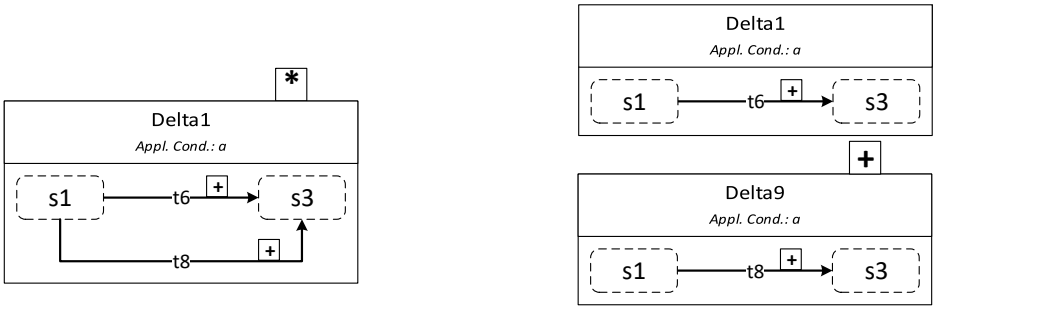
$$\alpha_{175}(e_{am}, \varphi(e_{am})) = \alpha_{175}(e_{am}, \varphi(e_{am}))^{\theta_u = \theta_i}.$$

Nachdem nun die Regeln für die Transformation von aus dem Delta-Modell entfernten Deltas hergeleitet wurden, werden im folgenden Abschnitt Regeln für modifizierte Deltas ermittelt.

Modifizierte Deltas

Für die Analyse der modifizierten Deltas wird davon ausgegangen, dass die Modifikation der folgenden Deltas in der neu hinzugefügten Version θ_3 erfolgt. Die Modifikation von Deltas kann auf eine vielfältige Weise erfolgen. So können etwa Operationen auf Elementen zum Delta hinzugefügt werden, das heißt, durch das Delta könnte in Zukunft ein Element hinzugefügt (beziehungsweise entfernt oder modifiziert) werden, welches vorher nicht durch das Delta hinzugefügt wurde. Ebenso können vor der Modifikation im Delta enthaltene Operationen entfernt werden. Weiterhin kann die Anwendungsbedingung oder die Anwendungsreihenfolge des Deltas abgeändert werden.

Hinzugefügte Operationen



(a) Modifiziertes Delta1

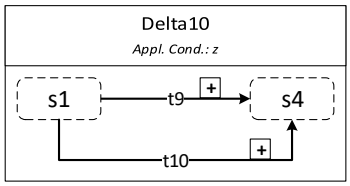
(b) Alternative Umsetzung des modifizierten Delta1

Abbildung 4.11.: Transformation eines modifizierten Deltas mit hinzugefügter Operation

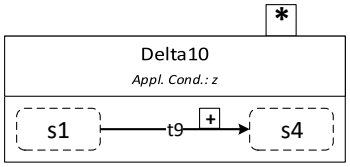
Das Hinzufügen von Operationen auf Elementen zum Delta führt dazu, dass ab der Version, in der das Delta modifiziert wird, diese Operationen neu existieren. Dies ließe sich ebenso erreichen, wenn das Delta nicht modifiziert wird und stattdessen ein neues Deltas zum Delta-Modell hinzugefügt wird, welches die neuen Operationen ausführt. Dies ist dargestellt in Abbildung 4.11. Abbildung 4.11a zeigt dabei das modifizierte Delta Delta1, welches zuvor nur Transition t6 zum Kernmodell hinzugefügt hat und nun zusätzlich Transition t8 hinzufügt. Abbildung 4.11b zeigt die alternative Umsetzung, bei der Delta1 nicht verändert, sondern stattdessen Delta9 zum Delta-Modell hinzugefügt wird, welches t8 einfügt. Für beide Möglichkeiten ergeben sich die gleichen Annotationen

für die im Delta beziehungsweise in den Deltas enthaltenen Elemente. Daher können Operationen, die durch eine Modifikation neu zu einem bestehenden Delta hinzugefügt werden, behandelt werden wie Operationen in einem hinzugefügten Delta. Dabei ist es völlig egal, um welche Art von Operation (hinzufügen, entfernen oder modifizieren) es sich handelt. Bei den bereits zuvor enthaltenen und nicht veränderten Elementen ergibt sich keine Änderung an der Annotation.

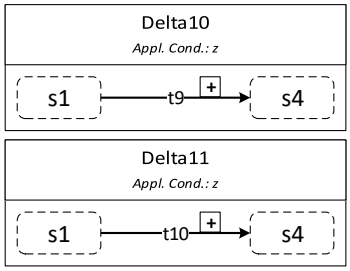
Entfernte Operationen



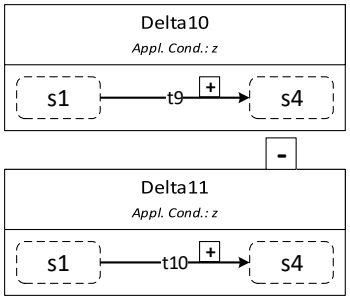
(a) Delta zum Hinzufügen von `t9` und `t10`



(b) Modifiziertes Delta mit entfernter Operation



(c) Alternative Umsetzung von `Delta10`



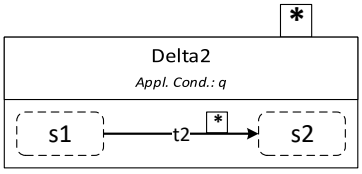
(d) Alternative Umsetzung des modifizierten `Delta10`

Abbildung 4.12.: Transformation eines modifizierten Deltas mit entfernter Operation

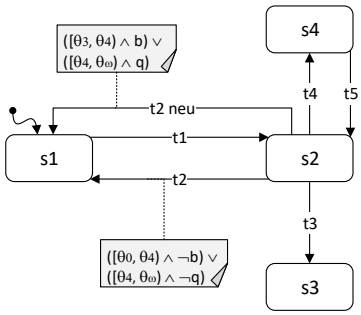
Das gleiche Prinzip trifft auf Operationen zu, die aus einem Delta entfernt werden. Zwar kann das Entfernen der Operationen in diesem Fall nicht durch das Entfernen eines anderen Deltas erreicht werden, der Effekt ist allerdings der gleiche. Sowohl das Entfernen von Operationen aus einem Delta als auch das Entfernen eines kompletten Deltas führt nämlich dazu, dass insgesamt Operationen aus dem Delta-Modell entfernt werden. Dies lässt sich an Abbildung 4.12 erkennen. Abbildung 4.12a zeigt hierbei das Delta `Delta10` vor der Modifikation. `Delta10` enthält zu diesem Zeitpunkt noch zwei Add-Operationen, einmal für Transition `t9` und einmal für `t10`. Nach der Modifikation, dargestellt in Abbildung 4.12b, enthält `Delta10` nur noch die Add-Operation für `t9`. Abbildung 4.12c zeigt eine alternative Möglichkeit, durch die sich `Delta10` ebenfalls umsetzen lässt, nämlich in Form von zwei getrennten Deltas, die jeweils eine Transition einfügen. Soll Transition `t10` nun ab einer Version nicht mehr eingefügt werden, kann einfach - wie in Abbildung 4.12d abgebildet - das Delta `Delta11` entfernt werden, welches `t10` einfügt. Dementsprechend können Operationen, die durch eine Modifikation aus einem bestehenden Delta entfernt werden, gehandhabt werden wie Operationen in einem entfernten Delta. Auch hierbei ist es wiederum vollkommen egal, um welche Art von Operation es sich dabei genau handelt.

Modifizierte Anwendungsreihenfolge oder -bedingung

Das Modifizieren der Anwendungsreihenfolge eines Deltas hat keine Auswirkungen auf die Annotationen des 175%-Modells, da die Anwendungsreihenfolge keine Änderung an der Gültigkeit eines Elements bewirkt, sondern lediglich zur Sicherstellung der Konsistenz des Delta-Modells dient. Eine Veränderung der Gültigkeit eines Elements würde ausschließlich dann resultieren, wenn etwa ein Element durch Änderung der Reihenfolge in Zukunft zuerst modifiziert wird, bevor es eingefügt wird. Hierbei handelt es sich allerdings um eine Modifikation, welche zeitgleich die Konsistenz des Delta-Modells stört. Eine solche Änderung ist daher ebenfalls verboten. Folglich nehmen alle erlaubten Modifikationen, welche die Konsistenz des Modells wahren, keine Änderungen an den Annotationen der Elemente vor.



(a) Delta mit modifizierter Anwendungsbedingung



(b) Annotation für `t2`

Abbildung 4.13.: Transformation eines Deltas mit modifizierter Anwendungsbedingung

Die Anwendungsbedingung eines Deltas wird geändert, indem die alte Anwendungsbedingung entfernt und stattdessen durch eine neue Bedingung ersetzt wird. Abbildung 4.13a zeigt `Delta2`, welches in Version θ_4 modifiziert wurde, sodass die vorherige Anwendungsbedingung b durch die neue Bedingung q ersetzt wurde. Um diese Modifikation im 175%-Modell umzusetzen, wird das letzte Intervall in den Annotationen aller Elemente, die im Delta enthalten sind, mit der aktuellen Version abgeschlossen. Darüber hinaus werden über eine Oder-Verknüpfung ein neues Intervall von der aktuellen Version θ_i bis zur Version θ_ω sowie die neue aussagenlogische Formel

$$\gamma = \gamma^{-(\neg\psi_{bm})} \wedge \neg\psi_{am}$$

für entfernte oder zu modifizierende Elemente oder

$$\gamma = \gamma^{-(\psi_{bm})} \wedge \psi_{am}$$

für hinzugefügte oder modifizierte Elemente zur bisherigen Annotation hinzugefügt. ψ_{bm} ist die Anwendungsbedingung des Deltas vor der Modifikation, während ψ_{am} die Bedingung des Deltas nach der Modifikation ist. Je nach Operation auf dem Element wird für die neue Formel also die alte (negierte) Anwendungsbedingung aus der alten Formel γ entfernt und die neue (negierte) Anwendungsbedingung über eine Und-Verknüpfung angehängt.

Beispiel 9. *Für Transition* `t2` *lautet die alte Annotation* $([\theta_0, \theta_\omega) \wedge \neg b)$*. Die vorherige Feature-Bedingung ist somit* $\gamma = \neg b$*, beziehungsweise vollständig* $\gamma = true \wedge \neg b$*. Die alte Anwendungsbedingung von* `Delta2`

ist $\psi_{bm} = b$. Daher ergibt sich $\gamma^{-(\neg\psi_{bm})} = true$ als alte Feature-Bedingung, aus der die negierte, vorherige Anwendungsbedingung entfernt wurde. Die neue Anwendungsbedingung von `Delta2` *lautet $\psi_{am} = q$. Entsprechend der Formel für zu modifizierende Elemente resultiert daher $\gamma = true \wedge \neg q$, beziehungsweise verkürzt $\gamma = \neg q$, als neue Feature-Bedingung für* `t2`*.*

Für die originale Transition `t2` aus `Delta2` ergibt sich somit die Annotation $([\theta_0, \theta_4) \wedge \neg b) \vee ([\theta_4, \theta_\omega) \wedge \neg q)$, wie in Abbildung 4.13b dargestellt. Für die modifizierte Transition `t2neu` resultiert hingegen $([\theta_3, \theta_4) \wedge b) \vee ([\theta_4, \theta_\omega) \wedge q)$ als neue Annotation. Als allgemeine Formel ergibt sich daher für hinzugefügte oder modifizierte Elemente e die Annotation

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma^{-(\psi_{bm})} \wedge \psi_{am}).$$

Für entfernte oder zu modifizierende Elemente e resultiert

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma^{-(\neg\psi_{bm})} \wedge \neg\psi_{am})$$

als allgemeine Form der Annotation.

Mit Hilfe der allgemeinen Regeln, die in den letzten Abschnitten für die Annotationen erarbeitet wurden, lässt sich ein Algorithmus erstellen, der Higher-Order Delta-Modelle in 175%-Modelle umwandelt. Dieser Algorithmus wird im nächsten Abschnitt vorgestellt.

4.2. Algorithmus zur Transformation

Der Algorithmus zur Umwandlung von Higher-Order Delta-Modellen in 175%-Modelle basiert auf den Regeln, die in Kapitel 4.1 aufgestellt wurden. Die Regeln definieren genau, auf welche Weise die Operationen in einer Annotation wieder gegeben werden. Bei jeder Veränderung am Delta-Modell, die Auswirkungen auf ein Element e hat, wird die Annotation von e im 175%-Modell verändert. Eine Veränderung am Delta-Modell hat immer dann eine Auswirkung auf ein Element e, wenn ein Delta hinzugefügt, entfernt oder modifiziert wird, welches eine Operation auf e enthält. In diesem Fall wird die Gültigkeit des letzten Tupels der Annotation abgeschlossen, sofern sie zuvor noch nicht abgeschlossen war, und wenn erforderlich ein neues Tupel mit einem neuen Intervall und einer neuen oder veränderten aussagenlogischen Formel angehängt.

Neben den bereits definierten Regeln müssen für einen Algorithmus die Eingabe, die Ausgabe und eine Reihenfolge, in der die Deltas abgearbeitet werden, definiert werden. Da bei diesem Algorithmus ein Higher-Order Delta-Modell in ein 175%-Modell transformiert werden soll, ergibt sich als Eingabe ein Higher-Order Delta-Modell und als Ausgabe ein 175%-Modell. Ein Higher-Order Delta-Modell besteht dabei aus einem Kernmodell und einer Menge von Higher-Order Deltas, die jeweils aus einer Menge von Operationen auf Deltas bestehen. Darüber hinaus werden keine weiteren Eingaben benötigt, da die Informationen für die Annotationen direkt aus den Deltas und den angewendeten Operationen entnommen werden. Es wird hierbei vorausgesetzt, dass es sich bei der Eingabe um ein konfliktfreies, gültiges Higher-Order Delta-Modell handelt. Ein Hinzuziehen von weiteren Informationsquellen, wie bespielsweise Feature-Modellen, zur Sicherstellung der Konfliktfreiheit des resultierenden Modells ist daher nicht notwendig. Die Annahme ist, dass bei Eingabe eines fehler- und konfliktfreien Higher-Order Delta-Modells die Anwendung von korrekten Regeln wiederum zur Ausgabe eines fehler- und konfliktfreien 175%-Modells führt. Diese Annahme soll in Kapitel 4.3 bewiesen werden.

Des Weiteren muss eine Reihenfolge definiert werden, in denen die Higher-Order Deltas, die darin enthaltenen Operationen auf Deltas und die wiederum in den Deltas enthaltenen Operationen auf Elementen abgearbeitet werden. Da die sequentiell geordneten Higher-Order Deltas aufeinander aufbauen und sich jeweils auf das Delta-Modell beziehen, welches sich durch Anwendung ihres Vorgängers ergibt, wird an dieser Stelle definiert, dass die Higher-Order Deltas in ihrer Anwendungsreihenfolge betrachtet werden. Die Anwendungsreihenfolge der Higher-Order Deltas ist durch die Versionen definiert, in denen sie zum Higher-Order Delta-Modell hinzugefügt werden.

Reihenfolge der Delta-Betrachtung

Ein Higher-Order Delta kann eine beliebige Menge von hinzugefügten, entfernten und modifizierten Deltas enthalten. Intuitiv könnte man meinen, dass hierbei zuerst die hinzugefügten, dann die modifizierten und zuletzt die entfernten Deltas betrachtet werden, um sicherzugehen, dass ein Delta überhaupt existiert, bevor es modifiziert oder entfernt wird, oder dass es nicht entfernt wird, bevor es modifiziert wird. Allerdings kann davon ausgegangen werden, dass ein Delta nicht in derselben Version hinzugefügt und wieder entfernt oder modifiziert wird. In diesem Fall könnte das Delta nämlich stattdessen entweder gar nicht erst eingefügt oder direkt in der veränderten Variante eingefügt werden, da die hinzugefügte Variante niemals verwendet werden kann. Auch Modifizieren und Entfernen in einem Schritt kann ausgeschlossen werden, da es auch hier genügt, das Delta einfach zu entfernen, da die modifizierte Version nicht verwendet werden kann. Eine sichere Reihenfolge der Deltas, bei der die Deltas zuerst hinzugefügt und erst als letztes entfernt werden, ist somit schon durch die Reihenfolge der Higher-Order Deltas abgedeckt und braucht innerhalb eines Higher-Order Deltas nicht beachtet zu werden. Daher werden bei der Bestimmung einer Reihenfolge für die Betrachtung der Deltas direkt die enthaltenen Operationen mit einbezogen.

Operationen auf Deltas und Operationen auf Elementen werden im Folgenden als Kombinationen $deltaOp(elementOp(element))$ betrachtet, wobei $deltaOp = (\mathsf{add}/\mathsf{rem}/\mathsf{mod}/\mathsf{mod}_{\mathsf{add}}/\mathsf{mod}_{\mathsf{rem}})$, $elementOp = (\mathsf{add}/\mathsf{rem}/\mathsf{mod}/\varepsilon)$ und $element = (e/cond)$. Die Variable *element* kann also ein Element e oder eine Delta-Anwendungsbedingung *cond* sein. Eine Element-Operation *elementOp* kann den Wert ε annehmen. Das bedeutet, dass keine Operation auf einem Element ausgeführt wird. Dies ist der Fall, wenn $element = cond$, da in diesem Fall nur eine Delta-Operation, jedoch keine Element-Operation angewendet wird, da kein Element betroffen ist. Eine Add-Operation in einem hinzugefügten Delta wird folglich als $\mathsf{add}(\mathsf{add}(e))$-Kombination bezeichnet, während eine Add-Operation in einem entfernten Delta als $\mathsf{rem}(\mathsf{add}(e))$-Kombination bezeichnet wird. Eine Modifikation eines Deltas, die eine Remove-Operation zum Delta hinzufügt, wird repräsentiert durch eine $\mathsf{mod}_{\mathsf{add}}(\mathsf{rem}(e))$-Kombination, und eine Modifikation eines Deltas, die eine Modify-Operation aus einem Delta entfernt, wird dargestellt als $\mathsf{mod}_{\mathsf{rem}}(\mathsf{mod}(e))$-Kombination. Die Modifikation einer Anwendungsbedingung wird bezeichnet als $\mathsf{mod}(cond)$-Kombination.

Die einzige Kombination aus Delta-Operation und Element-Operation, mit deren Regel keine vorher bestehende Annotation durch die Anweisung $\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ abgeschlossen wird, ist die $\mathsf{add}(\mathsf{add}(e))$-Kombination. Daraus kann gefolgert werden, dass diese Kombination als einzige nicht auf vorherige Operationen auf dem Element angewiesen ist. Aus diesem Grund muss diese Kombination in einem Higher-Order Delta immer zuerst betrachtet werden, da sie die grundlegende Voraussetzung für jede andere Kombination schafft, indem sie eine neue Annotation bei einem Element öffnet. Folglich müssten hinzugefügte Deltas zuerst betrachtet werden.

Die einzige Kombination, die kein neues Intervall beginnt, ist die $\mathsf{rem}(\mathsf{add}(e))$-Kombination. Diese schließt stattdessen die Annotation vorläufig endgültig ab. Da, wie soeben erläutert, alle Kombinationen außer der $\mathsf{add}(\mathsf{add}(e))$-Kombination auf eine nicht abgeschlossene Annotation angewiesen sind, ergibt sich daraus, dass die $\mathsf{rem}(\mathsf{add}(e))$-Kombination erst zum Schluss innerhalb eines Higher-Order Deltas betrachtet werden darf. Folglich müssten entfernte Deltas als letztes betrachtet werden. Modifizierte Deltas müssten dementsprechend nach den hinzugefügten und vor den entfernten Deltas ausgewertet werden.

An dieser Stelle ergibt sich allerdings ein Problem. $\mathsf{mod}_{\mathsf{add}}(\mathsf{add}(e))$-Kombinationen erzeugen die gleiche Annotation wie $\mathsf{add}(\mathsf{add}(e))$-Kombinationen. Sie eröffnen somit ebenfalls ein Intervall und sind daher genau wie letztere vor jeder anderen Kombination und somit noch vor den $\mathsf{add}(\mathsf{mod}(e))$- und $\mathsf{add}(\mathsf{rem}(e))$-Kombinationen einzuordnen. Anderenfalls könnte in einem hinzugefügten Delta ein Element entfernt oder modifiziert werden, das erst in einem modifizierten Delta eingefügt wird. In diesem Fall würde versucht werden, ein geöffnetes Intervall der Annotation für das Element abzuschließen, obwohl die Annotation kein geöffnetes Intervall besitzt. Im schlimmsten Fall existiert das Element noch überhaupt nicht im 175%-Modell, obwohl bereits versucht wird, es zu modifizieren.

Beispiel 10. *In einem Delta-Modell existiert eine Menge von Deltas, darunter Delta δ_M. In Version θ_i modifiziert Higher-Order Delta δ_i^H das Delta δ_M, sodass es zusätzlich eine Add-Operation für Element e_m enthält. Element e_m wird an dieser Stelle zum ersten Mal in der Evolutionshistorie betrachtet und besitzt somit noch keinerlei Annotation. Darüber hinaus fügt δ_i^H ein neues Delta δ_N zum Delta-Modell hinzu. δ_N modifiziert Element e_m. Werden zuerst die hinzugefügten Deltas ausgewertet, versucht der Algorithmus die bestehende Annotation von e_m durch die Regel $\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \neg\psi))$ zu erweitern. Element e_m existiert zu diesem Zeitpunkt allerding noch nicht im 175%-Modell und besitzt folglich auch noch keine Annotation, die erweitert werden kann. Bevor das hinzugefügte δ_N ausgewertet werden kann, muss also das modifizierte δ_M ausgewertet werden, welches e_m zum 175%-Modell hinzufügt und eine geöffnete Annotation erstellt.*

Aus diesem Grund muss die $\mathsf{mod}_{\mathsf{add}}(\mathsf{add}(e))$-Kombination vor $\mathsf{add}(\mathsf{rem}(e))$- und $\mathsf{add}(\mathsf{mod}(e))$-Kombinationen betrachtet werden. Allerdings muss auch die $\mathsf{add}(\mathsf{add}(e))$-Kombination weiterhin vor den $\mathsf{add}(\mathsf{rem}(e))$- und $\mathsf{add}(\mathsf{mod}(e))$-Kombinationen betrachtet werden. Bei einer bloßen Sortierung nach der Art der Delta-Operation ergeben sich also Konflikte bei der Anwendung der Algorithmus-Regeln. Die Element-Operationen dürfen daher überhaupt nicht deltaweise betrachtet werden. Vielmehr muss eine Reihenfolge anhand der Kombinationen aus Delta-Operation und Element-Operation festgelegt werden.

Zur Erstellung dieser Reihenfolge erhalten die unterschiedlichen Kombinationen Prioritäten. Durch Kategorisierung der verschiedenen Kombinationen ergeben sich insgesamt die Prioriäten 1 bis 5. 1 ist dabei die höchste Priorität, sodass diese Kombinationen als erstes betrachtet werden müssen. 5 besitzt die geringste Priorität, wodurch die betroffenen Kombinationen als letztes betrachtet werden. Die Prioritäten werden anhand der Art der Kombination automatisch vergeben. Die soeben erörterten Kombinationen $\mathsf{add}(\mathsf{add}(e))$ und $\mathsf{mod}_{\mathsf{add}}(\mathsf{add}(e))$ erhalten beide Priorität 1.

Neben diesen beiden Kombinationen annotieren außerdem die Kombinationen $\mathsf{add}(\mathsf{mod}(e))$ und $\mathsf{mod}_{\mathsf{add}}(\mathsf{mod}(e))$ Elemente, die entweder keine oder eine abgeschlossene Annotation besitzen. Diese Elemente sind die Nachher-Versionen der von der Modify-Operation betroffenen Elemente.

Folglich müssen auch diese Kombinationen vor allen anderen Kombinationen ausgewertet werden. Weiterhin dürfen diese Kombinationen aber erst nach den Kombinationen mit Priorität 1 ausgewertet werden, da sichergestellt sein muss, dass e schon existiert. Die beiden Kombinationen erhalten daher Priorität 2.

Wie weiter oben erläutert, beginnt $\mathsf{rem}(\mathsf{add}(e))$ als einzige Kombination kein neues Intervall und darf somit erst nach allen anderen Kombinationen betrachtet werden. Diese Kombination erhält somit Priorität 5. Da die Kombination $\mathsf{mod}_{\mathsf{rem}}(\mathsf{add}(e))$ die Annotation nach der gleichen Regel bildet wie Kombination $\mathsf{rem}(\mathsf{add}(e))$, erhält auch $\mathsf{mod}_{\mathsf{rem}}(\mathsf{add}(e))$ die Priorität 5.

Des Weiteren schließen auch die Kombinationen $\mathsf{rem}(\mathsf{mod}(e))$ und $\mathsf{mod}_{\mathsf{rem}}(\mathsf{mod}(e))$ Intervalle von Annotationen ab, ohne ein neues Intervall zu beginnen. Dies ist der Fall bei den Annotationen der Nachher-Versionen der von der Modify-Operation betroffenen Elemente. Aus diesem Grund müssen auch diese Kombinationen nach allen anderen Kombinationen betrachtet werden. Allerdings müssen die beiden Kombinationen noch vor den Kombinationen mit der Priorität 5 ausgewertet werden, da die Vorher-Versionen der von der Modify-Operation betroffenen Elemente auf einem geöffneten Intervall aufbauen. Es muss somit sichergestellt werden, dass die Intervalle dieser Elemente in jedem Fall erst danach durch die Kombinationen mit Priorität 5 abgeschlossen werden können. Die Kombinationen $\mathsf{rem}(\mathsf{mod}(e))$ und $\mathsf{mod}_{\mathsf{rem}}(\mathsf{mod}(e))$ erhalten somit die Priorität 4.

Die verbliebenen Kombinationen $\mathsf{add}(\mathsf{rem}(e))$, $\mathsf{mod}_{\mathsf{add}}(\mathsf{rem}(e))$, $\mathsf{mod}(\mathit{cond})$, $\mathsf{mod}_{\mathsf{rem}}(\mathsf{rem}(e))$ und $\mathsf{rem}(\mathsf{rem}(e))$ schließen alle ein bereits geöffnetes Intervall ab und eröffnen gleichzeitig ein neues Intervall. Diese Kombinationen müssen dementsprechend nach den Kombinationen mit Priorität 1 und 2 sowie vor den Kombinationen mit Priorität 4 und 5 ausgewertet werden. Sie erhalten daher Priorität 3, wodurch sich insgesamt fünf verschiedene Prioritäten ergeben, nach denen die Kombinationen geordnet werden müssen.

Zusammengefasst ergibt sich für die verschiedenen Kombinationen also die Betrachtungsreihenfolge

1. $\mathsf{add}(\mathsf{add}(e))$, $\mathsf{mod}_{\mathsf{add}}(\mathsf{add}(e))$
2. $\mathsf{add}(\mathsf{mod}(e))$, $\mathsf{mod}_{\mathsf{add}}(\mathsf{mod}(e))$
3. $\mathsf{add}(\mathsf{rem}(e))$, $\mathsf{mod}_{\mathsf{add}}(\mathsf{rem}(e))$, $\mathsf{mod}(\mathit{cond})$, $\mathsf{mod}_{\mathsf{rem}}(\mathsf{rem}(e))$, $\mathsf{rem}(\mathsf{rem}(e))$
4. $\mathsf{rem}(\mathsf{mod}(e))$, $\mathsf{mod}_{\mathsf{rem}}(\mathsf{mod}(e))$
5. $\mathsf{rem}(\mathsf{add}(e))$, $\mathsf{mod}_{\mathsf{rem}}(\mathsf{add}(e))$

An dieser Stelle wird also berücksichtigt, dass alle Operationen, die sich nicht auf vorherige geöffnete Intervalle beziehen, zuerst betrachtet werden. Hierbei werden normale Elemente vor den Nachher-Versionen von modifizierten Elementen eingeordnet. Dann folgen alle Operationen, die ein offenes Intervall abschließen und ein neues Intervall starten. Danach kommen die Operationen, die Intervalle von Nachher-Versionen von modifizierten Elementen abschließen, ohne ein neues Intervall zu öffnen. Abschließend werden alle Operationen betrachtet, die Intervalle von normalen Elementen abschließen, ohne ein neues Intervall zu öffnen.

Beschreibung des Algorithmus

Nachdem nun die Regeln für die Annotation von Elementen, die Betrachtungsreihenfolge für alle Kombinationen aus Delta-Operation und Element-Operation sowie die Ein- und Ausgaben defi-

niert wurden, ist in Algorithmus 4.1 der Algorithmus zur Transformation von Higher-Order Delta-Modellen in 175%-Modelle beschrieben.

Der Algorithmus erstellt zunächst jedes Element, das im Kernmodell sm_{core} des eingegebenen Higher-Order Delta-Modells enthalten ist, als temporal annotiertes Element im neuen 175%-Modell sm_{175} (Zeile 2) und annotiert das Element entsprechend der Annotationsregel für Kernelemente ($\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge true$) (Zeile 3). Darauf folgt das Erstellen und Ordnen der Kombinationen aus Delta-Operation und Element-Operation für jedes Higher-Order Delta δ^H aus der Menge Δ^H der Higher-Order Deltas des HOD-Modells (Zeile 5). Jede Kombination enthält dabei ein Element, die Anwendungsbedingung des Deltas, durch welches das Element verändert wird, sowie die auf das Element angewendete Kombination aus Delta-Operation und Element-Operation.

Algorithmus 4.1 .: Algorithmus zur Transformation von HODs in 175%-Modelle

```
   Input : HOD model (sm_core, Δ^H)
   Output : 175% model (sm_175, α_175)
1  forall e ∈ sm_core do
2      create(e, sm_175);
3      annotate(e, θ_0, true);
4  forall δ^H ∈ Δ^H do
5      combinations = orderCombinations(δ^H);
6      forall c ∈ combinations do
7          if e ∈ sm_175 then
8              annotate(e, θ_i, ψ);
9          else
10             create(e, sm_175);
11             annotate(e, θ_i, ψ);
```

Nach Erstellung der Kombinationen für ein Higher-Order Delta δ^H, werden die Kombinationen nach Priorität abgearbeitet und für jede Kombination geprüft, ob das enthaltene Elemente bereits im 175%-Modell existiert (Zeile 7). Falls das Element bereits im 175%-Modell existiert, wird es entsprechend der Annotationsregel für die jeweilige Kombination aus den in Kapitel 4.1 definierten Regeln mit den Parametern für die durch δ^H beschriebene Version θ_i und die Delta-Anwendungsbedingung ψ annotiert (Zeile 8). Existiert das Element noch nicht im 175%-Modell, wird es zunächst als temporal annotiertes Element zum 175%-Modell hinzugefügt (Zeile 10) und nachfolgend nach dem gleichen Prinzip annotiert wie die bereits exitierenden Elemente (Zeile 11).

Im folgenden Kapitel soll nun der Beweis erbracht werden, dass der beschriebene Algorithmus bei Eingabe eines korrekten Higher-Order Delta-Modells ein äquivalentes 175%-Modell ausgibt.

4.3. Beweis des Transformationsalgorithmus

Der Beweis für die Korrektheit des Algorithmus 4.1 soll mithilfe der mathematischen Induktion erfolgen. Der Algorithmus gilt dann als korrekt, wenn er ein 175%-Modell erzeugt, das äquivalent zum eingegebenen Higher-Order Delta-Modell ist. Der Beweis basiert auf der Annahme, dass ein

Higher-Order Delta-Modell und ein 175%-Modell genau dann äquivalent sind, wenn für jede Version der Evolutionshistorie das zugehörige Delta-Modell beziehungsweise 150%-Modell, das durch das Higher-Order Delta-Modell beziehungsweise das 175%-Modell repräsentiert werden, äquivalent sind. Unter welchen Aspekten ein Delta-Modell und ein 150%-Modell äquivalent sind, wird im Folgenden erörtert.

4.3.1. Äquivalenz Delta-Modell und 150%-Modell

Delta-Modellierung und 150%-Modellierung sind zwei unterschiedliche, etablierte Modellierungstechniken [35], um verschiedene Produktmodelle einer Softwareproduktlinie durch Variabilitätsmodelle zusammenzufassen und zu modellieren. In beiden Ansätzen ist eindeutig definiert, wie sich Produktmodelle aus den Variabilitätsmodellen ableiten lassen.

Um herleiten zu können, unter welchen Umständen ein 150%-Modell und ein Delta-Modell äquivalent sind, wird ein Feature-Modell FM mit einer Feature-Menge $F_{FM} = \{f_1, ..., f_n\}$ betrachtet. Für dieses Feature-Modell gilt, dass

$$\forall f \in F_{FM} : \exists \Gamma_f = \{f \mapsto true\}.$$

Für jedes Feature $f \in F_{FM}$ existiert also eine Feature-Konfiguration Γ_f, bei der Feature f selektiert ist. Daraus folgt, dass ebenfalls ein Produkt P_f existiert, das f enthält, und somit auch ein Produktmodell PM_f, welches das Produkt P_f repräsentiert.

Bei einem gegebenen, korrekten 150%-Modell muss nach Definition jedes Element des Modells mindestens in einem Produktmodell enthalten sein, wobei durch die Annotationen der Elemente eingeschränkt wird, welches Element in welchen Produktmodellen enthalten ist [12, 14]. Die einzelnen Elemente sind dazu mit den Features annotiert, für die sie gelten, beziehungsweise mit den Features, für die sie nicht gelten. Damit ein Element e im Produktmodell PM_f enthalten ist, muss es folglich mit f annotiert sein. Umgekehrt ist ein Element e, das mit f annotiert ist, automatisch im Produktmodell PM_f enthalten. Des Weiteren ist e dementsprechend nicht im Produktmodell PM_f erhalten, wenn es mit $\neg f$ annotiert ist.

Die Annotation von e mit f kann dabei sowohl einzeln als auch innerhalb einer aussagenlogischen Formel $\mathcal{F} \in \mathbb{B}(F_{FM})$ über mehrere Feature-Parameter bestehen. Ist ein Element e mit einer beliebigen Formel $\mathcal{F}$ annotiert, dann ist es in jedem Produktmodell PM_Γ enthalten, für welches gilt, dass $\Gamma \vDash \mathcal{F}$. Die Formel $\mathcal{F}$ muss also durch die Feature-Konfiguration Γ erfüllt werden. Durch gezieltes Ausblenden aller Modellelemente, deren annotierte Formel $\mathcal{F}$ durch eine gegebene Feature-Konfiguration Γ nicht erfüllt wird, resultiert für Γ genau ein spezifisches Produktmodell PM_Γ. Die Menge an Produktmodellen, die sich aus einem gegebenen 150%-Modell ableiten lässt, ist somit durch die Menge der möglichen Feature-Konfigurationen des zugeordneten Feature-Modells FM vordefiniert.

Bei einem gegebenen, korrekten Delta-Modell muss ebenfalls nach Definition jedes Element des Modells, das entweder im Kernmodell oder in einem Delta vorkommt, mindestens in einem Produktmodell enthalten sein, wobei durch spezifische Mengen von Deltas festgelegt wird, welches Element in welchen Produktmodellen enthalten ist [10, 34]. Die angewendete Menge von Deltas wird über die Anwendungsbedingung ψ_δ der Deltas bestimmt. Die Bedingung ψ_δ besteht dabei genauso wie die Annotationen im 150%-Modell in Form einer aussagenlogischen Formel $\mathcal{F}$. Um ein Produktmodell PM_Γ zu erhalten, werden alle Deltas δ angewendet, für die $\Gamma \vDash \psi_\delta$ gilt.

Folglich ist ein Element e genau dann im Produktmodell PM_Γ enthalten, wenn es im Kernmodell enthalten ist und nicht durch ein Delta mit ψ_δ entfernt wird oder wenn es nicht im Kernmodell enthalten ist, aber durch ein Delta mit ψ_δ hinzugefügt wird. Im Gegenzug ist ein Element e dementsprechend genau dann nicht im Produktmodell PM_Γ enthalten, wenn es im Kernmodell enthalten ist und durch ein Delta mit ψ_δ entfernt wird oder wenn es nicht im Kernmodell enthalten ist und auch nicht durch ein Delta mit ψ_δ hinzugefügt wird. In allen Fällen gilt an dieser Stelle $\Gamma \vDash \psi_\delta$. Durch die Anwendung aller Deltas, deren Anwendungsbedingung ψ_δ durch eine gegebene Feature-Konfiguration Γ erfüllt wird, resultiert für Γ also genau ein spezifisches Produktmodell PM_Γ. Auch hier ist also die Menge an Produktmodellen, die sich aus einem gegebenen Delta-Modell ableiten lässt, durch die Menge der möglichen Feature-Konfigurationen des zugeordneten Feature-Modells vordefiniert.

Ein Delta-Modell und ein 150%-Modell sind genau dann äquivalent, wenn sich aus ihnen für jede mögliche Feature-Konfiguration die gleichen Produktmodelle ableiten lassen. Die Bedeutung der Äquivalenz von Variabilitätsmodellen ist in Definition 2 zusammengefasst. Betrachtet werden zwei beliebige Modelle aus der Menge $\mathcal{M}_\mathcal{V} \setminus \mathcal{M}_\mathcal{E} \subset \mathcal{M}$, wobei $\mathcal{M}$ das Universum aller möglichen Modelle, $\mathcal{M}_\mathcal{V} \subset \mathcal{M}$ das Universum aller möglichen Variabilitätsmodelle und $\mathcal{M}_\mathcal{E} \subset \mathcal{M}_\mathcal{V}$ das Universum aller möglichen evolvierenden Variabilitätmodelle beschreibt. Folglich bezeichnet $\mathcal{M}_\mathcal{V} \setminus \mathcal{M}_\mathcal{E}$ das Universum aller Variabilitätsmodelle ohne die Menge der evolvierenden Variabilitätsmodelle. Ein evolvierendes Variabilitätsmodell bezeichnet ein Variabilitätsmodell, das zusätzlich Evolutionsinformationen einschließt, wie beispielsweise ein 175%-Modell oder ein Higher-Order Delta-Modell.

Definition 2. *Zwei Modelle $M_i, M_j \in \mathcal{M}_\mathcal{V} \setminus \mathcal{M}_\mathcal{E}$ sind äquivalent im Sinne $M_i \sim M_j$, genau dann, wenn sich für jede gegebene Feature-Konfiguration Γ aus M_i und M_j zwei Produktmodelle $PM_\Gamma^{M_i}$ und $PM_\Gamma^{M_j}$ ableiten lassen, sodass gilt $PM_\Gamma^{M_i} = PM_\Gamma^{M_j}$.*

Ein Delta-Modell und ein 150%-Modell für dieselbe Softwareproduktlinie sind folglich immer äquivalent, da durch die möglichen Feature-Konfigurationen der Produktlinie die jeweiligen Produktmodelle bereits vordefiniert sind. Basierend auf dieser Definition wird in Kapitel 4.3.2 die Äquivalenz von evolvierenden Modellen erläutert.

4.3.2. Äquivalenz Higher-Order Delta-Modell und 175%-Modell

Bei der Higher-Order Delta-Modellierung lassen sich nach Definition die verschiedenen Versionen des zugrundeliegenden Delta-Modells ableiten, indem die gegebenen Higher-Order Deltas sukzessiv auf das Kern-Delta-Modell angewendet werden [25]. Im Gegenzug lassen sich bei der 175%-Modellierung nach Definition die verschiedenen Versionen des zugrundeliegenden 150%-Modells ableiten, indem alle Elemente des 175%-Modells ausgeblendet werden, deren Annotation kein Tupel mit der entsprechenden Version enthält (s. Kapitel 3.2). Ergeben sich bei der Ableitung für jede Version θ ein Delta-Modell DM_θ und ein 150%-Modell (M, α_{150}^θ), für die gilt, dass $DM_\theta \sim (M, \alpha_{150}^\theta)$, da sie für die gleiche Produktlinie definiert sind, dann sind auch Higher-Order Delta-Modell und 175%-Modell äquivalent. Dies ist der Fall, da sich dann aus beiden Modellen bei Eingabe der gleichen Version θ und der gleichen Feature-Konfiguration Γ das gleiche Produktmodell PM_Γ ableiten lässt. Dementsprechend ergibt sich die allgemeine Definition der Äquivalenz von evolvierenden Variabilitätsmodellen in Definition 3.

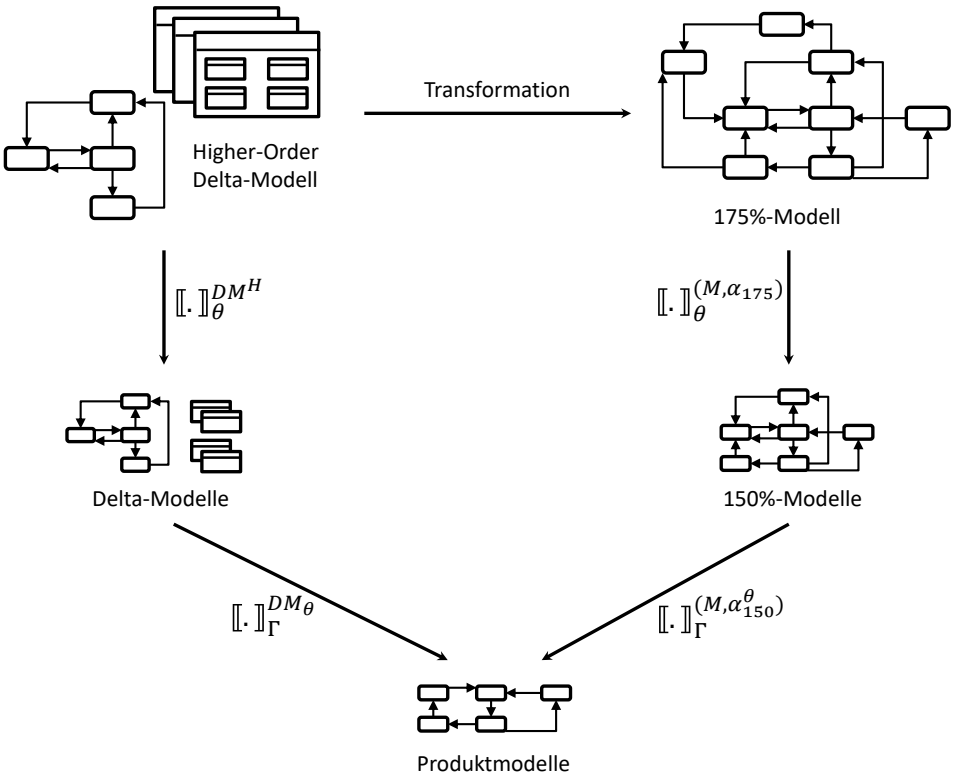


Abbildung 4.14.: Produktableitung von Higher-Order Delta- und 175%-Modellen

Definition 3. *Zwei evolvierende Variabilitätsmodelle $M_i, M_j \in \mathcal{M}_\mathcal{E}$ für eine Versionshistorie mit der Versionsmenge $\Theta = \{\theta_0, ..., \theta_n\}$ sind äquivalent im Sinne $M_i \sim_e M_j$, genau dann, wenn sich für jede gegebene Version $\theta \in \Theta$ aus M_i und M_j zwei Modelle $M_{i\theta}$ und M_{j_θ} ableiten lassen, sodass gilt $M_{i\theta} \sim M_{j_\theta}$.*

Folglich sind ein Higher-Order Delta-Modell und ein 175%-Modell für dieselbe Softwareproduktlinie immer äquivalent, da die Produktmodelle für jede Version θ eindeutig durch die möglichen Feature-Konfigurationen von θ vorgegeben sind. Die ableitbaren Delta- und 150%-Modelle sind also für jede Version äquivalent und somit sind es auch Higher-Order Delta-Modell und 175%-Modell.

Damit der Algorithmus 4.1 korrekt ist, muss daher auch das vom Algorithmus ausgegebene 175%-Modell entsprechend Definition 3 äquivalent zum eingegebenen Higher-Order Delta-Modell sein, da beide Modelle die gleiche Softwareproduktlinie repräsentieren sollen. Dies ist genau dann der Fall, wenn sich aus dem erzeugten 175%-Modell nur 150%-Modelle ableiten lassen, die äquivalent zu den ableitbaren Delta-Modellen des eingegebenen Higher-Order Delta-Modells sind. Dieser Zusammenhang ist in Abbildung 4.14 dargestellt.

Für eine Softwareproduktlinie SPL bezeichnet $[\![.]\!]_\theta^{DM^H}$ die Funktion, mit der aus einem Higher-Order Delta-Modell DM^H für jede Version θ der Versionsmenge Θ_{SPL} der Evolutionshistorie der Softwareproduktlinie ein Delta-Modell DM_θ abgeleitet wird. Dementsprechend ist $[\![.]\!]_\theta^{(M,\alpha_{175})}$ die Funktion, mit der aus einem 175%-Modell (M, α_{175}) für jede Version $\theta \in \Theta_{SPL}$ ein 150%-Modell (M, α_{150}^θ) abgeleitet wird. Folglich charakterisiert $[\![.]\!]_\Gamma^{DM_\theta}$ die Funktion, die für jede gültige Feature-Konfiguration Γ der Produktlinie SPL aus einem Delta-Modell DM_θ ein Produktmodell PM_Γ ableitet. Ebenso leitet die Funktion $[\![.]\!]_\Gamma^{(M,\alpha_{150}^\theta)}$ für jede Feature-Konfiguration Γ aus einem 150%-Modell

$(M, \alpha^{\theta}_{150})$ ein Produktmodell PM_{Γ} ab. Da es sich sowohl bei der Ableitung aus DM^{H} als auch aus (M, α_{175}) um die gleiche Produktlinie mit der gleichen Versionsmenge Θ als auch der gleichen Menge an Feature-Konfigurationen Γ handelt, resultiert aus beiden evolvierenden Modellen die gleiche Menge an Produktmodellen PM_{Γ}.

Theorem 1. *Da für ein Higher-Order Delta-Modell DM^{H} und ein 175%-Modell (M, α_{175}) derselben Softwareproduktlinie immer gilt, dass $DM^{H} \sim_{e} (M, \alpha_{175})$, ist der vorgestellte Algorithmus 4.1 genau dann korrekt, wenn für das vom Algorithmus erzeugte 175%-Modell $(M, \alpha_{175})_{A}$ und das eingegebene Higher-Order Delta-Modell DM^{H}_{A} immer gilt, dass $(M, \alpha_{175})_{A} \sim_{e} DM^{H}_{A}$. Bei einer Evolutionhistorie $\Theta = \{\theta_0, ..., \theta_n\}$ gilt, dass $(M, \alpha_{175})_{A} \sim_{e} DM^{H}_{A}$, wenn gilt, dass $\forall \theta \in \Theta : DM_{\theta} \sim (M, \alpha^{\theta}_{150})$.*

Nach Theorem 1 ist der Algorithmus folglich korrekt, wenn bewiesen werden kann, dass das erzeugte 175%-Modell auch tatsächlich immer äquivalent zum eingegebenen Higher-Order Delta-Modell ist. Durch die folgende mathematische Induktion soll diese Äquivalenz bewiesen werden. Dazu wird gezeigt, dass sich für jede Version ein 150%-Modell aus dem 175%-Modell ableiten lässt, das äquivalent ist zum Delta-Modell, das für dieselbe Version aus dem Higher-Order Delta-Modell abgeleitet wird.

4.3.3. Induktionsbeweis

Der Algorithmus fügt alle Elemente des Kernmodells vom Higher-Order Delta-Modell zum 175%-Modell hinzu. Danach betrachtet er nacheinander jedes Higher-Order Delta δ^{H} und die Elemente, die durch eins der Deltas verändert werden, auf denen δ^{H} eine Operation ausführt. Stößt der Algorithmus dabei auf ein Element, für das gilt $e \neq e_{exists}$ für alle $e_{exists} \in E_{175}$, dann wird e zu E_{175} und somit zum 175%-Modell hinzugefügt. Dabei gilt, dass $e = e_{exists}$, wenn e und e_{exists} in sämtlichen Eigenschaften gleich sind. Somit werden auch die Versionen e_{bm} und e_{am} eines modifizierten Elements e im 175%-Modell durch zwei unterschiedliche Elemente repräsentiert, da sie sich durch die Modifikation in mindestens einer Eigenschaft unterscheiden. Sobald ein Element im 175%-Modell enthalten ist, wird es nicht wieder vom Algorithmus entfernt.

Der Algorithmus liefert folglich ein 175%-Modell, dass sämtliche Elemente enthält, die bis zu der betrachteten Version auch im Higher-Order Delta-Modell vorkommen. Da die Existenz aller Elemente im Higher-Order Delta-Modell somit durch den Algorithmus garantiert ist, folgt, dass der Algorithmus ein zum HOD-Modell äquivalentes 175%-Modell erstellt, wenn sich durch die in Kapitel 4.1 definierten Regeln die korrekten Annotationen für die Elemente ergeben. Eine Annotation für ein Element e ist korrekt, wenn sie für jede Version θ ein Tupel enthält, welches der Annotation des 150%-Modells entspricht, das äquivalent ist zum Delta-Modell für Version θ, welches durch das Higher-Order Delta-Modell repräsentiert wird.

Induktionsanfang

Für den Beweis der Korrektheit des Algorithmus muss zunächst gezeigt werden, dass das erzeugte 175%-Modell für ein konkretes n äquivalent zum eingegebenen Higher-Order Delta-Modell ist. Daher wird für den Induktionsanfang $n = 0$ gewählt. Für den Beweis der Äquivalenz des erzeugten 175%-Modells, wird ein Higher-Order Delta-Modell DM^{H} für eine Evolutionshistorie $\Theta_n = \{\theta_0, ..., \theta_n\}$ betrachtet. Durch $n = 0$ umfasst die Historie zunächst nur eine Version θ_0, sodass gilt $\Theta_0 = \{\theta_0\}$. DM^{H} besteht an dieser Stelle nur aus dem Kern des Higher-Order Delta-Modells. Dieser Kern ist ein Delta-Modell mit Kernmodell M_{core} und einer Menge Δ von initialen Deltas δ.

Das HOD-Modell DM^H für $\Theta_0 = \{\theta_0\}$ repräsentiert somit zu Beginn das Delta-Modell DM_{θ_0} für Version θ_0.

Ein 175%-Modell (M, α_{175}) für eine Evolutionshistorie mit Versionen $\Theta_0 = \{\theta_0\}$ entspricht nach Definition einem 150%-Modell (s. Kapitel 3). Dieses 150%-Modell muss nach Definition 3 äquivalent zum Delta-Modell DM_{θ_0} sein, damit das Higher-Order Delta-Modell DM^H und das 175%-Modell für Θ_0 nach Definition 3 äquivalent sind.

Die Erzeugung von DM_{θ_0} durch den Algorithmus erfolgt in zwei Schritten. Im ersten Schritt werden nur die Elemente des Kernmodells von DM_{θ_0}, welches ein gültiges Produktmodell repräsentiert, in das 175%-Modell überführt. Im zweiten Schritt werden die Deltas von DM_{θ_0} betrachtet und deren Einschränkungen oder Erweiterungen in das 175%-Modell eingearbeitet.

Die Elemente des Kernmodells von DM_{θ_0} werden durch den Algorithmus zum 175%-Modell hinzugefügt und mit der Formel $\alpha_{175}(e, \varphi(e)) = [\theta_0, \theta_\omega) \wedge true$ annotiert. Da die Evolutionshistorie $\Theta_n = \{\theta_0, ..., \theta_n\}$ zunächst nur die Menge $\Theta_0 = \{\theta_0\}$ umfasst, gilt außerdem:

$$\Theta_0 = \{\theta_0\} \Rightarrow [\theta_0, \theta_\omega) = \{\theta_0\} \tag{4.1}$$

Dadurch folgt, dass $\alpha_{175}(e, \varphi(e)) = \theta_0 \wedge true$. Da ausschließlich θ_0 betrachtet wird, ist θ_0 mit *true* belegt, sodass gilt $\alpha_{175}(e, \varphi(e)) = true \wedge true$. Jedes Element e ist folglich für jede Feature-Konfiguration im Produktmodell enthalten, sodass alle im 175%-Modell enthaltenen Elemente zusammen das in diesem Schritt bisher einzig mögliche Produktmodell bilden.

Da im ersten Schritt nur das Kernmodell von DM_{θ_0} generiert wird, werden die initialen Deltas und die dadurch entstehenden Einschränkungen und Erweiterungen zunächst außen vor gelassen. Es wird quasi ein nicht valides Delta-Modell ohne jegliche Deltas betrachtet. Das einzige Produktmodell, dass ohne eine Anwendung von Deltas aus dem Kernmodell resultieren kann, ist somit das Kernmodell selbst. Das Kernmodell beschreibt an dieser Stelle für das nicht valide Delta-Modell also das einzig mögliche Produktmodell, welches für jede beliebige Feature-Konfiguration resultiert. Da sämtliche Elemente des Kernmodells dem 175%-Modell hinzugefügt wurden, wodurch Kernmodell und 175%-Modell somit dieselbe Menge an Produktmodellen - nämlich nur das Kernmodell selbst - für jede beliebige Feature-Konfiguration erzeugen, sind Higher-Order Delta-Modell, in diesem Schritt nur bestehend aus dem Kernmodell von DM_{θ_0}, und 175%-Modell bis hierhin nach Definition 3 äquivalent. □

Im zweiten Schritt werden die Deltas von DM_{θ_0} miteinbezogen. Hierbei sind drei Fälle zu unterscheiden: Hinzufügen, Entfernen oder Modifizieren eines Elements.

Hinzugefügte Elemente Im ersten Fall wird ein Delta δ betrachtet, das ein Element e unter der Bedingung ψ_δ zum Produktmodell hinzufügt. Damit lassen sich nun zwei unterschiedliche Produktmodelle aus dem Delta-Modell ableiten: ein Produktmodell, das e enthält, und ein Produktmodell, das e nicht enthält. Entsprechend Kapitel 4.3.1 ergibt sich das Produktmodell, das e enthält, für jede Feature-Konfiguration Γ, für die gilt $\Gamma \vDash \psi_\delta$. Umgekehrt resultiert das Produktmodell, das e nicht enthält, für jede Feature-Konfiguration Γ, für die gilt $\Gamma \vDash \neg\psi_\delta$.

Durch den Algorithmus wird e zum 175%-Modell hinzugefügt. Die durch den Algorithmus angewendete Regel für diesen Fall lautet $\alpha_{175}(e, \{\theta_0\}) = [\theta_0, \theta_\omega) \wedge \psi$. Aufgrund von Gleichung 4.1 gilt somit $\alpha_{175}(e, \{\theta_0\}) = \theta_0 \wedge \psi$. Nach Definition ergibt sich im ableitbaren 150%-Modell von θ_0 für e die Annotation $\alpha^{\theta_0}_{150}(e) = \psi$. Folglich lassen sich aus dem 150%-Modell nun zwei unterschiedliche Produktmodelle ableiten. Eins der Produktmodelle enthält e und ergibt sich nach Kapitel 4.3.1 für

jede Feature-Konfiguration Γ, für die $\Gamma \models \psi$ gilt, während das andere Produktmodell e nicht enthält und sich für jede Feature-Konfiguration Γ ableiten lässt, für die $\Gamma \models \neg\psi$ gilt.

Entfernte Elemente In diesem Fall wird ein Delta δ betrachtet, das ein Element e unter der Bedingung ψ_δ aus einem Produktmodell entfernt. Folglich lassen sich zwei unterschiedliche Produktmodelle aus dem Delta-Modell ableiten. Das Produktmodell ohne Element e resultiert für jede Feature-Konfiguration Γ, für die gilt $\Gamma \models \psi_\delta$, während das Produktmodell mit e für jede Feature-Konfiguration Γ abgeleitet wird, für die $\Gamma \models \neg\psi_\delta$ gilt.

Der Transformationsalgorithmus fügt e zum 175%-Modell hinzu. Die Regel zur Annotation lautet in diesem Fall $\alpha_{175}(e, \{\theta_0\}) = [\theta_0, \theta_\omega) \wedge (\gamma \wedge \neg\psi_\delta)$. Der Ausdruck γ bezeichnet die Annotation von e, die zuvor bestand. Analog zu hinzugefügten Elementen ergibt sich im ableitbaren 150%-Modell von θ_0 für e die Annotation $\alpha^{\theta_0}_{150}(e) = \gamma \wedge \neg\psi_\delta$. Aus dem 150%-Modell lassen sich daher nun zwei unterschiedliche Produktmodelle ableiten, nämlich eines das e enthält und eines das e nicht enthält. Das Produktmodell, das e nicht enthält, ergibt sich für jede Feature-Konfiguration Γ, für die $\Gamma \models \psi_\delta$ gilt, und das Produktmodell, das e enthält, resultiert für jede Feature-Konfiguration Γ, für die $\Gamma \models \neg\psi_\delta$ gilt.

Modifizierte Elemente Im letzten Fall wird ein Delta δ betrachtet, das ein Element e unter der Bedingung ψ_δ im Produktmodell modifiziert. An dieser Stelle müssen die beiden Möglichkeiten unterschieden werden, dass e entweder bereits im Kernmodell enthalten ist oder dass e erst durch ein Delta δ' mit $\psi_{\delta'} = \gamma$ zum Produktmodell hinzugefügt wird. Ist e im Kernmodell enthalten, können zwei unterschiedliche Produktmodelle aus dem Delta-Modell abgeleitet werden. Eines enthält e_{bm}, die Version von e vor der Modifikation, während das andere e_{am}, die Version von e nach der Modifikation, enthält. Ersteres resultiert für jede Feature-Konfiguration Γ, für die $\Gamma \models \neg\psi_\delta$ gilt, und letzteres lässt sich für jede Konfiguration Γ mit $\Gamma \models \psi_\delta$ ableiten. Wird e erst durch δ' hinzugefügt, ergibt sich zusätzlich die Möglichkeit, dass e nicht im Produktmodell enthalten ist. Dies trifft für jedes Γ mit $\Gamma \models \neg\gamma$ zu. Im Umkehrschluss können e_{bm} und e_{am} nur für Konfigurationen im Produktmodell enthalten sein, für die $\Gamma \models \gamma$ gilt. Folglich lässt sich in diesem Fall das Produktmodell mit e_{bm} für jede Konfiguration Γ mit $\Gamma \models \gamma \wedge \neg\psi_\delta$ ableiten. Im Gegenzug resultiert das Produktmodell mit e_{am} für jede Konfiguration Γ, für die $\Gamma \models \gamma \wedge \psi_\delta$ gilt.

Element e_{bm} existiert bereits zuvor im 175%-Modell, dabei bestehen zwei verschiedene Möglichkeiten. Entweder ist e_{bm} mit *true* annotiert oder es ist mit einer beliebigen Formel γ annotiert. Element e_{am} hingegen wird durch den Algorithmus neu zum 175%-Modell hinzugefügt. Die Annotation für e_{bm} lautet nach den Regeln des Algorithmus $\alpha_{175}(e_{bm}, \{\theta_0\}) = [\theta_0, \theta_\omega) \wedge (\gamma \wedge \neg\psi_\delta)$. Als 150%-Annotation ergibt sich folglich die Annotation $\alpha^{\theta_0}_{150}(e_{bm}) = \gamma \wedge \neg\psi_\delta$. Für e_{am} resultiert die Annotation $\alpha_{175}(e_{am}, \{\theta_0\}) = [\theta_0, \theta_\omega) \wedge (\gamma_{e_{bm}} \wedge \psi_\delta))$ und somit die 150%-Annotation $\alpha^{\theta_0}_{150}(e_{am}) = \gamma_{e_{bm}} \wedge \psi_\delta$. War e_{bm} zuvor mit *true* annotiert, können aus dem 150%-Modell für θ_0 daher nun zwei unterschiedliche Produktmodelle abgeleitet werden, wobei eines e_{bm} enthält, während das andere e_{am} enthält. Element e_{bm} ist im Produktmodell für jede Feature-Konfiguration Γ enthalten, für die $\Gamma \models \neg\psi_\delta$ gilt. Hingegen ist e_{am} im Produktmodell enthalten, das für jede Feature-Konfiguration Γ ableitbar ist, für die $\Gamma \models \psi_\delta$ gilt. War e_{bm} allerdings zuvor mit $\gamma_{e_{bm}}$ annotiert, ergeben sich drei unterschiedliche Produktmodelle, wobei eines e_{bm}, aber nicht e_{am} enthält, und ein weiteres e_{am}, aber nicht e_{bm} enthält, während das letzte weder e_{bm} noch e_{am} enthält. Hierbei ist Element e_{bm} im Produktmodell für jede Feature-Konfiguration Γ enthalten, für die $\Gamma \models \gamma_{e_{bm}} \wedge \neg\psi_\delta$ gilt. Element e_{am} ist im Produktmodell

für jede Feature-Konfiguration Γ mit $\Gamma \vDash \gamma_{e_{bm}} \wedge \psi_{\delta}$ enthalten. Bei einer Feature-Konfiguration Γ, für die $\Gamma \vDash \neg\gamma_{e_{bm}}$ gilt, ist weder e_{bm} noch e_{am} im Produktmodell enthalten.

Folglich lassen sich für alle drei Fälle aus Delta-Modell und 150%-Modell für θ_0 die gleichen Produktmodelle für die gleichen Feature-Konfigurationen ableiten. Nach Definition 2 sind diese beiden somit äquivalent, woraus folgt, dass das Higher-Order Delta-Modell und das 175%-Modell nach Definition 3 ebenfalls äquivalent sind. Die Verwendung der Regeln für Kernelemente und initiale Deltas erzeugt somit ein zum Higher-Order Delta-Modell äquivalentes 175%-Modell. □

Induktionsvoraussetzung

Bis Schritt n gilt, dass das durch den Algorithmus erzeugte 175%-Modell äquivalent zum eingegebenen Higher-Order Delta-Modell ist. Weiterhin gilt bis Schritt n, dass $\alpha_{150}^{\theta_n}(e) = \alpha_{175}(e, \{\theta_n\})$.

Wenn das 175%-Modell bis Schritt n äquivalent zum Higher-Order Delta-Modell ist, dann gelten außerdem folgende Annotationen in den ableitbaren 150%-Modellen bis n:

- $\alpha_{150}^{\theta_n}(e) = \psi$ für hinzugefügte Elemente,
- $\alpha_{150}^{\theta_n}(e) = \gamma \wedge \neg\psi$ für entfernte Elemente und
- - $\alpha_{150}^{\theta_n}(e) = \gamma \wedge \neg\psi$ für Elemente e_{bm} beziehungsweise
  - $\alpha_{150}^{\theta_n}(e) = \gamma_{e_{bm}} \wedge \psi$ für Elemente e_{am}.

Induktionsschritt

Damit der Algorithmus korrekt ist, muss gezeigt werden, dass die Induktionsvoraussetzung auch für jedes weitere, beliebige $n + 1$ gilt. Daher wird in diesem Induktionsschritt die Äquivalenz des erzeugten 175%-Modells für $n + 1$ gezeigt.

Für die bisherige Evolutionshistorie Θ_n lässt sich aus dem Higher-Order Delta-Modell DM^H eine Menge von Delta-Modellen $DM_{\Theta} = \{DM_{\theta_0}, ..., DM_{\theta_n}\}$ ableiten. Das letzte Delta-Modell DM_{θ_n} besitzt eine Menge Δ_{θ_n} von Deltas. Die Menge $\Delta_{\theta_n}^{e} \subset \Delta_{\theta_n} = \{\delta_{\theta_n}^{e_0}, ..., \delta_{\theta_n}^{e_m}\}$ bezeichnet die Menge der Deltas $\delta_{\theta_n}^{e}$ von DM_{θ_n}, welche eine Operation auf dem Element e ausführen.

Die Evolutionshistorie Θ_n erweitert sich für $n + 1$ um eine Version auf $\Theta_{n+1} = \Theta_n \cup \theta_{n+1}$. Das Higher-Order Delta-Modell DM^H erhält für die Version θ_{n+1} ein Higher-Order Delta δ_{n+1}^{H}. Durch δ_{n+1}^{H} wird ein neues Delta-Modell $DM_{\theta_{n+1}}$ repräsentiert. Für den Beweis der verschiedenen Regeln aus Kapitel 4.1 in einem Induktionsschritt müssen unterschiedliche Fälle berücksichtigt werden:

1. Die Menge $\Delta_{\theta_n}^{e}$ ändert sich durch δ_{n+1}^{H} nicht, sodass $\Delta_{\theta_{n+1}}^{e} = \Delta_{\theta_n}^{e}$
2. Der Menge $\Delta_{\theta_n}^{e}$ wird durch δ_{n+1}^{H} ein neues Delta $\delta_{\theta_{n+1}}^{e_{m+1}}$ hinzugefügt, sodass $\Delta_{\theta_{n+1}}^{e} = \Delta_{\theta_n}^{e} \cup \delta_{\theta_{n+1}}^{e_{m+1}}$
3. Aus der Menge $\Delta_{\theta_n}^{e}$ wird durch δ_{n+1}^{H} ein bisher bestehendes Delta $\delta_{\theta_n}^{e_k}$ entfernt, sodass $\Delta_{\theta_{n+1}}^{e} = \Delta_{\theta_n}^{e} \setminus \{\delta_{\theta_n}^{e_k}\}$
4. In der Menge $\Delta_{\theta_n}^{e}$ wird durch δ_{n+1}^{H} ein bisher bestehendes Delta $\delta_{\theta_n}^{e_k}$ modifiziert, sodass $\Delta_{\theta_{n+1}}^{e} = \Delta_{\theta_n}^{e}$ mit $\delta_{\theta_n}^{e_k} \neq \delta_{\theta_{n+1}}^{e_k}$

Die Fälle 2 und 3 unterteilen sich dabei noch einmal in untergeordnete Fälle für die verschiedenen Element-Operationen Hinzufügen, Entfernen oder Modifizieren. Auch für die verschiedenen

Element-Operationen können teilweise noch einmal untergeordnete Fälle bestehen. Der Fall 4 unterteilt sich wiederum in das Hinzufügen und Entfernen von Element-Operationen sowie das Modifizieren der Anwendungsbedingung, wobei sich diese Fälle ebenfalls wieder weiter unterteilen lassen. Im Folgenden wird jeder der vier Fälle samt der verschiedenen Unterfälle in diesem Induktionsschritt einzeln gezeigt.

Fall 1

Für den Induktionsschritt wird zunächst Fall 1 betrachtet. Bei diesem Fall führt keines der Deltas, die durch δ^H_{n+1} hinzugefügt, entfernt oder modifiziert werden, eine Operation auf Element e aus. Sämtliche Deltas, die e beeinflussen, bleiben somit unverändert. Folglich ist e in Version θ_{n+1} in den gleichen Produktmodellen enthalten wie in Version θ_n. Somit muss im 150%-Modell für θ_{n+1} das Element e die gleiche Annotation haben wie im 150%-Modell für θ_n. Daraus folgt, dass

$$\alpha^{\theta_n}_{150}(e) = \alpha^{\theta_{n+1}}_{150}(e) \tag{4.2}$$

gelten muss, wenn kein Delta, das auf e operiert, verändert wird. Entsprechend der Regeln aus Kapitel 4.1 gilt für add e, dass

$$\alpha_{175}(e, \{\theta_n, \theta_{n+1}\}) = [\theta_n, \theta_\omega) \wedge \psi, \tag{4.3}$$

da keine Veränderung an der Annotation erfolgt, wenn kein hinzugefügtes, entferntes oder modifiziertes Delta eine Operation auf e ausführt. Für die Versionsmenge $\Theta_{n+1} = \{\theta_n, \theta_{n+1}\}$ entspricht

$$[\theta_n, \theta_\omega) = \{\theta_n, \theta_{n+1}\}. \tag{4.4}$$

Daraus folgt entsprechend der Definition von 175%-Modellen in Kapitel 3 für die Element-Operation add e (und analog für die Operationen rem e und mod e), dass:

$$[\theta_n, \theta_\omega) \wedge \psi = \{(\theta_n \wedge \psi), (\theta_{n+1} \wedge \psi)\}, \tag{4.5}$$

wobei gilt, dass

$$\alpha^{\theta_n}_{150}(e) = \theta_n \wedge \psi \tag{4.6}$$

und

$$\alpha^{\theta_{n+1}}_{150}(e) = \theta_{n+1} \wedge \psi. \tag{4.7}$$

Nach den Gleichungen 4.5, 4.6 und 4.7 gilt somit, dass

$$[\theta_n, \theta_\omega) \wedge \psi = \{\alpha^{\theta_n}_{150}(e), \alpha^{\theta_{n+1}}_{150}(e)\}. \tag{4.8}$$

Aufgrund von Gleichung 4.3 folgt daraus, dass

$$\alpha_{175}(e, \{\theta_n, \theta_{n+1}\}) = \{\alpha^{\theta_n}_{150}(e), \alpha^{\theta_{n+1}}_{150}(e)\}. \tag{4.9}$$

Aufgrund von 4.4 sind n und $n + 1$ mit *true* belegt, sodass aus

$$\theta_n \wedge \psi = \theta_{n+1} \wedge \psi \tag{4.10}$$

folgt, dass

$$true \wedge \psi = true \wedge \psi. \tag{4.11}$$

Durch die Gleichungen 4.6 und 4.7 ergibt sich Gleichung 4.2 als wahr. Die Algorithmus erzeugt folglich Annotationen $\alpha_{175}(e, \varphi(e))$, die für jede weitere Version Θ_{n+1} die gleiche $\alpha_{150}(e)$-Annotation enthalten, solange keine Änderung an den Deltas erfolgt, die auf e operieren. Das erzeugte 175%-Modell ist daher für Fall 1 für jede weitere Version Θ_{n+1} weiterhin äquivalent zum Higher-Order Delta-Modell. □

Fall 2

Als nächstes wird für den Induktionsschritt Fall 2 betrachtet. Bei diesem Fall werden durch δ^H_{n+1} Deltas hinzugefügt, die auf e operieren. Hierbei muss wieder durch Unterfälle unterschieden werden, ob diese hinzugefügten Deltas e hinzufügen, entfernen oder modifizieren. Sämtliche bisher bestehende Deltas, die e beeinflussen, bleiben unberührt.

Der Beweis für die Operationen add e und mod e für den Fall $\alpha_{175}(e, \varphi(e)) = \bot$ funktioniert analog zum Beweis für initiale Deltas. Der Ausdruck θ_0 in den Regeln wird für hinzugefügte Deltas nämlich lediglich durch θ_i ersetzt. Die anderen Fälle für diese Operationen werden einzeln gezeigt.

Hinzugefügte Elemente Durch δ^H_{n+1} wird zusätzlich ein Delta hinzugefügt, welches eine Add-Operation auf e ausführt. Bei den Add-Operationen muss eine Fallunterscheidung getroffen werden. Für den Fall $\alpha_{175}(e, \varphi(e)) \neq \bot \wedge \theta_u \leq \theta_i$ gilt die Regel

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e)) \vee ([\theta_i, \theta_\omega) \wedge \psi). \tag{4.12}$$

Im an dieser Stelle schon bestehenden 175%-Modell (M, α_{175}) existiert bereits ein Element e mit der Annotation $\alpha_{175}(e, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_n) \wedge \gamma$. Der Ausdruck $[\theta_m, \theta_n) \wedge \gamma$ bildet bereits eine abgeschlossene Annotation. Diese bleibt durch den Teilausdruck $\alpha_{175}(e, \varphi(e))$ der zu beweisenden Regel unberührt. Somit werden die bisher bestehenden Versionen $\{\theta_m, ..., \theta_n\}$, die durch (M, α_{175}) repräsentiert werden, nicht beeinflusst und die bisherige Äquivalenz der Modelle bleibt weiterhin bestehen. Dies ist korrekt, da sowohl die bisher von DM^H respräsentierten Delta-Modelle als auch die vom 175%-Modell repräsentierten 150%-Modelle für die verschiedenen Versionen von δ^H_{n+1} unbeeinflusst bleiben.

Stattdessen repräsentiert das Higher-Order Delta-Modell durch das neue Higher-Order Delta nun zusätzlich ein weiteres Delta-Modell für die Version θ_{n+1}. Damit das 175%-Modell nach Definition 3 weiterhin äquivalent zum HOD-Modell ist, muss auch das 175%-Modell ein zusätzliches 150%-Modell für Version θ_{n+1} respräsentieren, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Da das Element e in diesem Fall hinzugefügt wird, wird die Annotation im äquivalenten 150%-Modell für θ_{n+1} nach Voraussetzung durch $\alpha^{\theta_{n+1}}_{150}(e) = \psi$ gebildet. Dies entspricht, wie für initiale Deltas bereits bewiesen, dem Teilausdruck $[\theta_i, \theta_\omega) \wedge \psi$ aus der zu beweisenden Regel. Dies ist für diesen Fall korrekt, da alle vorherigen Intervalle der Annotation von e abgeschlossen sind und somit vorherige Feature-Bedingungen keinen Einfluss auf die neue Feature-Bedingung haben. Dementsprechend erweitert die Regel 4.12 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Für den Fall $\alpha_{175}(e, \varphi(e)) \neq \bot \wedge \theta_u = \theta_\omega$ gilt die Regel

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \psi)) \tag{4.13}$$

In diesem Fall existiert im 175%-Modell (M, α_{175}) das Element e diesmal bereits mit der Annotation $\alpha_{175}(e, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_\omega) \wedge \gamma$. Der Ausdruck $[\theta_m, \theta_\omega) \wedge \gamma$ bildet eine noch nicht abgeschlossene Annotation. Das Intervall enthält für die bisherige Versionsmenge Θ_n die Versionen $\{\theta_m, ..., \theta_n\}$. Die Annotation bleibt durch den Teilausdruck $\alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ unverändert bestehen, da hierbei lediglich das Argument θ_u des letzten Intervalls mit θ_{n+1} belegt wird, sodass $[\theta_i, \theta_{n+1})$. Da es sich um ein rechtsoffenes Intervall handelt, enthält das Intervall somit weiterhin die Versionen $\{\theta_i, ..., \theta_n\}$.

Für diesen Fall repräsentiert das Higher-Order Delta-Modell durch das neue Higher-Order Delta ebenfalls ein weiteres Delta-Modell für die Version θ_{n+1}. Zur Wahrung der Äquivalenz muss auch hier das 175%-Modell nun zusätzlich ein äquivalentes 150%-Modell für θ_{n+1} repräsentieren. Da das Element e in diesem Fall hinzugefügt wird, müsste die Annotation im äquivalenten 150%-Modell für θ_{n+1} durch $\alpha_{150}^{\theta_{n+1}}(e) = \psi$ gebildet werden. Allerdings besitzt e ein nicht abgeschlossenes Intervall, wodurch die diesem Intervall zugeordnete Feature-Bedingung auf keinen Fall vernachlässigt werden darf. Der Ausdruck γ in $\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \psi))$ bildet sich in der Form $\gamma = \gamma_1 \wedge ... \wedge \gamma_k$. Jedes Delta, das auf e operiert, ist dabei repräsentiert durch ein γ_j mit $\gamma_1 \leq \gamma_j \leq \gamma_k$. Hierbei bildet γ_j die (negierte) Anwendungsbedingung ψ beziehungsweise $\neg\psi$ des Deltas. Sämtliche Deltas, die auf e operieren, sind somit in γ vertreten.

Da durch δ_{n+1}^H nun ein weiteres Delta zum Delta-Modell für θ_{n+1} hinzugefügt wird, muss zu γ ein zusätzliches γ_{k+1} hinzugefügt werden, welches die Anwendungsbedingung für dieses Delta darstellt. Auf diese Weise wird ein äquivalentes 150%-Modell durch das 175%-Modell repräsentiert. Das Hinzufügen von γ_{k+1} geschieht durch den Ausdruck $\gamma \wedge \psi$, wobei $\gamma_{k+1} = \psi$. Dementsprechend erweitert die Regel 4.13 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Entfernte Elemente Durch δ_{n+1}^H wird ein Delta hinzugefügt, welches eine Remove-Operation auf e ausführt. Für die Operation rem e gilt die Regel

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma \wedge \neg\psi)). \tag{4.14}$$

Im bisher bestehenden 175%-Modell (M, α_{175}) existiert das Element e bereits mit der Annotation $\alpha_{175}(e, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_\omega) \wedge \gamma$.

Für den Teilausdruck $\alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ der zu beweisenden Regel funktioniert der Beweis analog zu den hinzugefügten Elementen. Wie zuvor repräsentiert das Higher-Order Delta-Modell durch das neue Higher-Order Delta zusätzlich ein weiteres Delta-Modell für die Version θ_{n+1}. Damit das 175%-Modell nach Definition 3 weiterhin äquivalent zum HOD-Modell ist, muss auch das 175%-Modell ein zusätzliches 150%-Modell für Version θ_{n+1} respräsentieren, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Da das Element e in diesem Fall entfernt wird, wird die Annotation im äquivalenten 150%-Modell für θ_{n+1} durch $\alpha_{150}^{\theta_{n+1}}(e) = \gamma \wedge \neg\psi$ gebildet. Dies entspricht, wie für initiale Deltas bereits bewiesen, dem Teilausdruck $[\theta_i, \theta_\omega) \wedge (\gamma \wedge \neg\psi)$ aus der zu beweisenden Regel. Dementsprechend erweitert die Regel 4.14 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Modifizierte Elemente Durch δ_{n+1}^H wird ein Delta hinzugefügt, welches eine Modify-Operation auf e ausführt. Für die Vorher-Version e_{bm} des Elements funktioniert der Beweis analog zum Beweis für

entfernte Elemente. Für die Nachher-Version e_{am} für den Fall $\alpha_{175}(e_{am}, \varphi(e_{am})) \neq \bot$ gilt die Regel

$$\alpha_{175}(e_{am}, \varphi(e_{am})) = \alpha_{175}(e_{am}, \varphi(e_{am})) \vee ([\theta_i, \theta_\omega) \wedge (\gamma_{e_{bm}} \wedge \psi)). \tag{4.15}$$

Im bisher bestehenden 175%-Modell (M, α_{175}) existiert das Element e_{am} bereits mit der Annotation $\alpha_{175}(e_{am}, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_n) \wedge \gamma$.

Genauso wie bei der Operation add e für den Fall $\alpha_{175}(e, \varphi(e)) \neq \bot \wedge \theta_u \leq \theta_i$ bildet $[\theta_m, \theta_n) \wedge \gamma$ bereits eine abgeschlossene Annotation für Element e. Diese Annotation bleibt hier analog durch den Teilausdruck $\alpha_{175}(e_{am}, \varphi(e_{am}))$ der zu beweisenden Regel unberührt und die bestehende Äquivalenz bleibt daher ebenso unangetastet. Daneben repräsentiert auch bei dieser Operation das Higher-Order Delta-Modell durch das neue Higher-Order Delta ein weiteres Delta-Modell für θ_{n+1}, wodurch das 175%-Modell wieder ebenso ein weiteres äquivalentes 150%-Modell repräsentieren muss.

Da e_{am} die Nachher-Version eines modifizierten Elements ist, wird die Annotation im äquivalenten 150%-Modell durch $\alpha_{150}^{\theta_{n+1}}(e_{am}) = \alpha_{150}(e_{bm}) \wedge \psi$ gebildet. Dies entspricht, wie für initiale Deltas bereits bewiesen, dem Teilausdruck $[\theta_i, \theta_\omega) \wedge (\gamma \wedge \psi)$ aus der zu beweisenden Regel. Dementsprechend erweitert die Regel 4.15 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Da sich aus dem 175%-Modell durch jede Regel der hinzugefügten Deltas ein zu $DM_{\theta_{n+1}}$ äquivalentes 150%-Modell ableiten lässt, ist das 175%-Modell nach Definition weiterhin äquivalent zum HOD-Modell. □

Fall 3

Weiterhin wird für den Induktionsschritt Fall 3 betrachtet. Bei diesem Fall werden durch δ_{n+1}^H Deltas entfernt, die Operationen auf e ausführen. Hierbei muss erneut durch Unterfälle unterschieden werden, ob die entfernten Deltas e hinzufügen, entfernen oder modifizieren.

Hinzugefügte Elemente Bei den Add-Operationen muss erneut eine Fallunterscheidung getroffen werden. Für den Fall $\gamma^+ \neq \psi$ gilt die Regel

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge \gamma^{-\psi} \tag{4.16}$$

Im Higher-Order Delta-Modell existieren bei Betrachtung dieses Falls temporär zwei Deltas, welche eine Add-Operation auf e ausführen. Durch δ_{n+1}^H wird das ältere dieser Deltas entfernt. Die restlichen, bestehenden Deltas, die e beeinflussen, bleiben unberührt. Im bisher bestehenden 175%-Modell (M, α_{175}) existiert das Element e mit der Annotation $\alpha_{175}(e, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_\omega) \wedge \gamma$.

Der Beweis funktioniert für den Teilausdruck $\alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ analog wie in den vorherigen Beweisen, die bisherige Äquivalenz bleibt also unverändert bestehen. Dazu wird durch das neue Higher-Order Delta auch hier ein neues Delta-Modell $DM_{\theta_{n+1}}$ für Version θ_{n+1} verkörpert. Dementsprechend muss das 175%-Modell ein zusätzliches, äquivalentes 150%-Modell repräsentieren.

Der Ausdruck γ in $\alpha_{175}(e, \{\theta_i, ..., \theta_n\}) = [\theta_i, \theta_\omega) \wedge \gamma$ bildet sich in der Form $\gamma = \gamma_1 \wedge ... \wedge \gamma_k$, wobei γ_j mit $1 \leq j \leq k$ jeweils die (negierte) Bedingung ψ oder $\neg\psi$ für ein Delta, welches auf e operiert, darstellt. Jedes Delta ist somit repräsentiert durch ein γ_j. Das Delta-Modell für θ_{n+1}, das durch das neue δ_{n+1}^H repräsentiert wird, verliert im Vergleich zum Delta-Modell DM_{θ_n} für Version θ_n ein Delta. Dementsprechend muss für die neue Feature-Bedingung für das äquivalente 150%-Modell für θ_{n+1} das entsprechende γ_j des Deltas aus dem alten γ entfernt werden. Somit verkörpert

der Ausdruck $\gamma^{-(\psi)}$ nach Definition korrekt nur noch die Bedingungen der anderen, weiterhin existierenden Deltas, die auf e operieren. Der Beweis für den Teilausdruck $[\theta_i, \theta_\omega)$ erfolgt wie zuvor. Dementsprechend erweitert die Regel 4.16 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zu $DM_{\theta_{n+1}}$ ist. Folglich ist das 175%-Modell nach Definition weiterhin äquivalent zum HOD-Modell. □

Für den Fall $\gamma^+ = \psi$ gilt die Regel

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \tag{4.17}$$

Im Higher-Order Delta-Modell existiert bei Betrachtung dieses Falls nur ein Delta, welches eine Add-Operation auf e ausführt. Über dieses Delta hinaus bestehen keine weiteren Deltas, die e beeinflussen. Durch δ^H_{n+1} wird das Delta entfernt, welches e hinzufügt. Im bisher bestehenden 175%-Modell (M, α_{175}) existiert das Element e mit der Annotation $\alpha_{175}(e, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_\omega) \wedge \psi$. Dies ist der Fall, da - aufgrund der Definition der Reihenfolge für den Algorithmus sowie durch die vorausgesetzte Konfliktfreiheit - alle Deltas, die auf e operieren, bereits entfernt sein müssen, wenn das einzige Delta entfernt wird, das e hinzufügt.

Durch den Ausdruck $\alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ wird nun das letzte Intervall abgeschlossen, die bisherige Äquivalenz bleibt somit bestehen. Darüber hinaus wird durch δ^H_{n+1} ein neues Delta-Modell $DM_{\theta_{n+1}}$ repräsentiert. Dementsprechend muss das 175%-Modell ein zusätzliches, äquivalentes 150%-Modell repräsentieren. Da das letzte Delta, das auf e operiert, durch δ^H_{n+1} entfernt wurde, ist e kein Bestandteil des Delta-Modells für θ_{n+1}. Folglich ist es auch kein Bestandteil eines äquivalenten 150%-Modells für θ_{n+1}. Dementsprechend besitzt e keine Annotation nach α_{150}. Da nach Voraussetzung $\alpha^{\theta_n}_{150}(e) = \alpha_{175}(e, \{\theta_n\})$ gilt, darf e ebenso keine Annotation für θ_{n+1} nach α_{175} besitzen. Da die zu beweisende Regel das bisherige Intervall mit θ_{n+1} abschließt, sodass $[\theta_i, \theta_{n+1}) = \{\theta_i, ..., \theta_n\}$, und darüber hinaus kein neues Intervall hinzufügt, besitzt $\alpha_{175}(e, \varphi(e))$ keine Annotation für θ_{n+1}. Dementsprechend erweitert die Regel 4.17 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zu $DM_{\theta_{n+1}}$ ist.

Entfernte Elemente Der Beweis für Remove-Operationen funktioniert analog zum Beweis für Add-Operationen für den Fall $\gamma^+ \neq \psi$. Hierbei muss der Ausdruck ψ lediglich durch den Ausdruck $\neg\psi$ ersetzt werden.

Modifizierte Elemente Der Beweis für Vorher-Versionen e_{bm} von Elementen aus Modify-Operationen funktioniert analog zum Beweis für Remove-Operationen. Der Beweis für Nachher-Versionen e_{am} funktioniert analog zum Beweis für Add-Operationen für den Fall $\gamma^+ = \psi$.

Durch jede Regel der entfernten Deltas lässt sich also aus dem 175%-Modell ein zu $DM_{\theta_{n+1}}$ äquivalentes 150%-Modell ableiten lässt. Folglich ist das 175%-Modell nach Definition weiterhin äquivalent zu DM^H. □

Fall 4

Als letztes wird für den Induktionsschritt Fall 4 untersucht. Bei diesem Fall werden durch δ^H_{n+1} bestehende Deltas modifiziert, die Operationen auf e ausführen. Auch für diesen Fall ergeben sich wieder Unterfälle. Hierbei wird zum einen unterschieden, ob Operationen zum Delta hinzugefügt oder entfernt werden. Zum anderen gilt es noch, die Regel für die Modifikation der Anwendungsbedingung eines Deltas zu beweisen.

Hinzugefügte und entfernte Operationen Der Beweis für Operationen, die durch die Modifikation eines Deltas zum Delta hinzugefügt werden, funktioniert analog zum Beweis für Operationen in hinzugefügten Deltas, da es sich um die gleichen Regeln handelt. Selbiges gilt für Operationen, die durch die Modifikation entfernt werden und entfernte Deltas.

Modifizierte Anwendungsbedingung Durch δ_{n+1}^{H} wird bei einem Delta, welches eine Operation auf e ausführt, die Anwendungsbedingung modifiziert. An dieser Stelle muss wieder eine Fallunterscheidung beachtet werden. Für Add-Operationen oder Nachher-Versionen von Elementen aus Modify-Operationen gilt die Regel

$$\alpha_{175}(e, \varphi(e)) = \alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i} \vee ([\theta_i, \theta_\omega) \wedge (\gamma^{-(\psi_{bm})} \wedge \psi_{am}). \tag{4.18}$$

Im 175%-Modell (M, α_{175}) existiert das Element e bereits mit der Annotation $\alpha_{175}(e, \{\theta_m, ..., \theta_n\}) = [\theta_m, \theta_\omega) \wedge \gamma$.

Für den Teilausdruck $\alpha_{175}(e, \varphi(e))^{\theta_u = \theta_i}$ funktioniert der Beweis analog wie zuvor. Durch ihn bleibt die bisherige Äquivalenz der Modelle bestehen. Der Teilausdruck $\gamma^{-(\psi_{bm})}$ kann wiederum analog zum Beweis für Add-Operationen für den Fall $\gamma^{+} \neq \psi$ in entfernten Deltas bewiesen werden. Dabei wird der Ausdruck γ_j mit $1 \leq j \leq k$ aus $\gamma = \gamma_1 \wedge ... \wedge \gamma_k$ entfernt, der die Anwendungsbedingung vor der Modifikation repräsentiert. Zuletzt wird der Beweis für der Teilausdruck ψ_{am} analog zum Beweis für Add-Operationen für den Fall $\alpha_{175}(e, \varphi(e)) \neq \bot \wedge \theta_u = \theta_\omega$ in hinzugefügten Deltas geführt. Hierbei wird der Ausdruck γ_{k+1}, der die Anwendungsbedingung nach der Modifikation repräsentiert, zum Ausdruck $\gamma^{-(\psi_{bm})}$ hinzugefügt. Dementsprechend erweitert die Regel 4.18 das 175%-Modell, sodass es zusätzlich ein 150%-Modell repräsentiert, das äquivalent zum Delta-Modell für θ_{n+1} ist.

Für Remove-Operationen und Vorher-Versionen von Elementen aus Modify-Operationen kann die Regel analog zur eben bewiesenen Regel bewiesen werden. An dieser Stelle werden lediglich die Teilausdrücke $\gamma^{-(\psi_{bm})}$ und ψ_{am} durch $\gamma^{-(\neg\psi_{bm})}$ und $\neg\psi_{am}$ ersetzt.

Somit kann durch jede Regel der modifizierten Deltas aus dem 175%-Modell ein zu $DM_{\theta_{n+1}}$ äquivalentes 150%-Modell abgeleitet werden. Das 175%-Modell ist dementsprechend nach Definition weiterhin äquivalent zu DM^{H}. □

Induktionsschluss

Die Regeln sind sowohl für die Kernelemente als auch für die initialen, hinzugefügten, entfernten und modifizierten Deltas korrekt. Durch sie wird für jede Version θ aus Θ die Annotation $\alpha_{175}(e, \varphi(e))$ so erweitert, dass durch sie ein 150%-Modell für θ ableitbar ist. Dieses 150%-Modell ist immer äquivalent zum Delta-Modell für Version θ, das aus dem Higher-Order Delta-Modell ableitbar ist, da beide die gleichen Produktmodelle für die selben Feature-Konfigurationen erzeugen. Nach Definition 3 gilt für das erzeugte 175%-Modell (M, α_{175}) und das eingegebene Higher-Order Delta-Modell DM^{H} somit, dass $(M, \alpha_{175}) \sim DM^{H}$. Entsprechend Theorem 1 folgt daher, dass Algorithmus 4.1 korrekt ist.

Im folgenden Kapitel wird die Implementierung von 175%-Modellen und dem in diesem Kapitel beschriebenen Transformationsalgorithmus vorgestellt.

5 Implementierung

Dieses Kapitel dokumentiert die Implementierung der Struktur von 175%-Modellen, des Transformationsalgorithmus zur Umwandlung von Higher-Order Delta- in 175%-Modelle und des Slicings von 175%-Modellen. Dazu werden zunächst die Anforderungen für diese Funktionalitäten beschrieben und anschließend die technische Realisierung erläutert. Da die Implementierung teilweise auf einer bestehenden Implementierung aufbaut, beziehungsweise eine bestehende Implementierung erweitert, wird im Zuge dessen erst die bereits bestehende Implementierung vorgestellt. Nachfolgend werden die umgesetzten Erweiterungen und Änderungen sowie Neuentwicklungen aufgeführt. Sowohl für die bestehende als auch die zusätzliche Implementierung werden dabei Datenmodelle und Architektur präsentiert.

5.1. Anforderungen

Die Anforderungen für die Funktionalitäten lassen sich in vier wesentliche Punkte unterteilen, für die sich wiederum kleinere Anforderungen ergeben. Das Use-Case-Diagramm in Abbildung 5.1 verbildlicht diese Anforderungen.

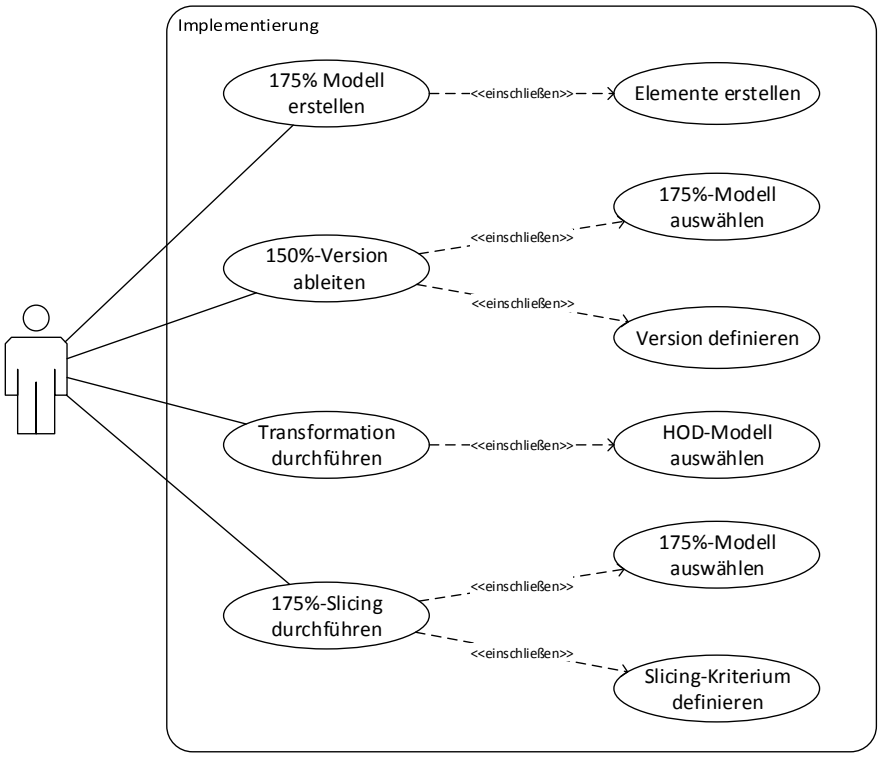


Abbildung 5.1.: Use-Case-Diagramm der Anforderungen

Zur Realisierung der Funktionalitäten soll der Benutzer die Möglichkeit haben, ein 175%-Modell erstellen und anzeigen zu können. Dieses Modell bietet gleichzeitig die Grundlage für das 175%-Slicing. Damit ein 175%-Modell angefertigt werden kann, müssen die einzelnen Elemente, aus denen das Modell besteht, per Hand definiert werden können, beispielsweise über einen Editor.

Zugleich soll die Möglichkeit bestehen, ein 175%-Modell automatisch aus einem bereits existierenden Higher-Order Delta-Modell generieren zu können. Dazu muss das Higher-Order Delta-Modell ausgewählt beziehungsweise angegeben werden können.

Sobald ein 175%-Modell vorliegt, indem es per Transformation erzeugt oder manuell erstellt wurde, soll es möglich sein, ein 150%-Modell für eine bestimmte Version aus dem 175%-Modell abzuleiten. Dazu muss der Benutzer sowohl eine Version als auch ein 175%-Modell auswählen können.

Schlussendlich soll auf einem existierenden 175%-Modell ein 175%-Slicing durchgeführt werden können. Zu diesem Zweck muss sowohl ein 175%-Modell angegeben werden können, als auch die Möglichkeit bestehen, ein Element, eine Feature-Konfiguration und eine Menge von Versionen für das Slicing-Kriterium zu definieren.

Zusätzlich sollte das Softwarewerkzeug leicht erweitert werden können, um ein eventuelles Hinzufügen von zusätzlichen Funktionalitäten zu erleichtern.

Insgesamt ergeben sich so mehrere Funktionen, die entweder einzeln oder als Gesamtpaket verwendet werden können. Abbildung 5.2 zeigt dazu einen Arbeitsablauf, der sich für eine kombinierte Nutzung der einzelnen Funktionen ergibt.

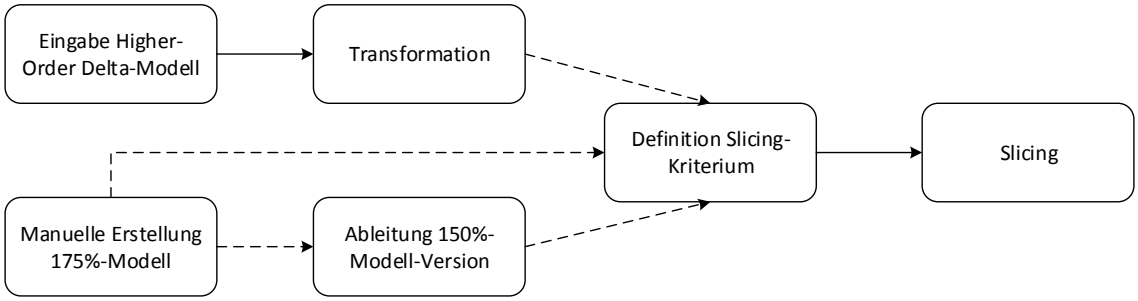


Abbildung 5.2.: Arbeitsablauf und Zusammenspiel der verschiedenen Aufgaben

Hierbei wird zuerst ein 175%-Modell entweder durch eine manuelle Anfertigung oder durch eine automatische Generierung aus einem Higher-Order Delta-Modell erstellt. Auf Wunsch kann dieses Modell dann durch Slicing auf bestimmte, wesentliche Elemente reduziert werden. Dazu kann entweder direkt ein 175%-Slicing durchgeführt werden, oder zunächst eine 150%-Modell-Version aus dem 175%-Modell abgeleitet und dann ein 150%-Modell-Slicing angewendet werden. Weiterhin kann eine 150%-Modell-Version auch abgeleitet werden, ohne danach ein Slicing durchführen zu müssen. Die gestrichelten Pfeile in Abbildung 5.2 bezeichnen folglich optionale Übergänge zu weiteren Schritten.

Nachdem nun die Anforderungen an das Werkzeug sowie ein Arbeitsablauf zur Einordnung der verschiedenen Funktionen definiert wurde, wird im folgenden Kapitel deren technische Realisierung beschrieben.

5.2. Realisierung

Einige der umzusetzenden Funktionen bauen auf bereits bestehenden Implementierungen auf, beziehungsweise erweitern diese. Daher stellt sich weder die Frage, ob das zu implementierende Werkzeug als eigenständige Software oder als Erweiterung einer bestehenden Entwicklungsumgebung umgesetzt werden soll, noch welche Entwicklungsumgebung hierfür konkret verwendet werden soll. Die bestehende Implementierung liegt in Form von Plug-ins für die Entwicklungsumgebung Eclipse[1] vor. Eclipse ist ein plattformunabhängiges Programmierwerkzeug zur Entwicklung von Software, welches unterschiedliche Programmiersprachen unterstützt. Dabei bietet Eclipse die Möglichkeit durch das Einbinden von selbst erstellbaren Plug-ins die Funktionalität der Entwicklungsumgebung zu erweitern. Um die bestehenden Plug-ins erweitern beziehungsweise mit neuen Plug-ins darauf aufbauen zu können, soll daher für die Implementierung ebenfalls Eclipse benutzt werden.

Zur Erstellung dieser Plug-ins soll das Eclipse Modeling Framework (EMF)[2] verwendet werden. Das Framework bietet umfangreiche Funktionen für eine Modell-basierte Entwicklung von Software. So können etwa auf Basis eines Meta-Modells automatisch Java-Klassen für die Klassen dieses Modells erzeugt werden. Weiterhin können direkt Adapterklassen zur Darstellung und Bearbeitung von Modellen geschaffen werden, die zugleich ein einfaches Generieren eines Editors für Modelle erlauben. Mit Hilfe dieses Frameworks können daher leicht Plug-ins auf der Basis eines Datenmodells entwickelt werden. Die bereits bestehende Implementierung in Form von Plug-ins basiert ebenfalls auf dem Eclipse Modeling Framework. Im Folgenden werden die Plug-ins mit den ihnen zugrunde liegenden Datenmodellen vorgestellt.

5.2.1. Datenmodelle der bestehenden Implementierung

In der als Plug-ins realisierten, bisher bestehenden Implementierung sind die jeweiligen Datenmodelle als EMF Meta-Modelle definiert. Auf den folgenden sieben Plug-ins baut die neue Implementierung im Wesentlichen auf:

- **de.imotep.core.datamodel:** Die Java-Klassen der Typen, welche von den Elementen eines Modells angenommen werden können
- **de.imotep.core.behavior:** Die Java-Klassen der Elemente einer normalen State Machine
- **de.imotep.variability.annotatedBehavior:** Die Java-Klassen der Elemente einer 150%-State Machine
- **de.imotep.variability.deltaBehavior:** Die Java-Klassen der Elemente eines Delta-Modells
- **de.imotep.dope:** Die Java-Klassen der Elemente eines Higher-Order Delta-Modells
- **de.imotep.slicing:** Die Java-Klassen der Elemente, welche zum Durchführen des Slicings von (150%-)State Machines benötigt werden
- **de.imotep.slicing.dependency:** Die Java-Klassen der Elemente, welche die Abhängigkeiten zwischen State Machine-Elementen beschreiben

[1] https://www.eclipse.org/ (Stand 15.10.2017)
[2] https://www.eclipse.org/modeling/emf/ (Stand 15.10.2017)

Einige dieser Plug-ins sind wiederum voneinander abhängig. Diese Abhängigkeiten werden in Kapitel 5.2.2 dargestellt, wenn das Zusammenspiel der neu implementierten Komponenten beschrieben wird.

de.imotep.core.datamodel Das Plug-in `de.imotep.core.datamodel` definiert die Grundlage vieler der anderen Plug-ins, in dem es Typen von Elementen spezifiziert, von denen die meisten Elemente der anderen Datenmodelle erben. In Abbildung 5.3 ist das Meta-Modell dieses Plug-ins abgebildet.

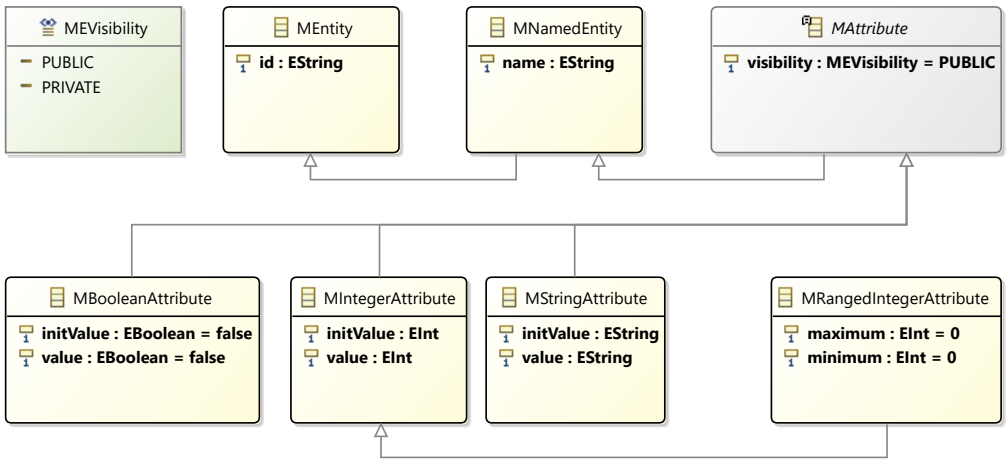


Abbildung 5.3.: Meta-Modell Plug-in `de.imotep.core.datamodel`

Ein Objekt der Klasse `MEntity` besitzt immer eine einzigartige ID, bei der abgeleiteten Klasse `MNamedEntity` muss zusätzlich ein Name vergeben werden. Von `MNamedEntity` erbt die abstrakte Klasse `MAttribute`, die darüber hinaus entweder öffentlich oder nur privat sichtbar (`visibility`) sein muss. `MAttribute` lässt sich wiederum durch die Klassen `MBooleanAttribute`, `MIntegerAttribute` und `MStringAttribute` näher spezifizieren, wobei `MIntegerAttribute` nochmal als `MRangedIntegerAttribute` abgeleitet werden kann.

de.imotep.core.behavior Im Plug-in `de.imotep.core.behavior` ist die Struktur einer State Machine definiert. Das zugehörige Meta-Modell ist in Abbildung 5.4 dargestellt. Sämtliche der enthaltenen Klassen sind von der Klasse `MBehaviorEntity` abgeleitet, die wiederum von der Klasse `MNamedEntity` des Plug-ins `de.imotep.core.datamodel` abgeleitet ist. Bei `MAttribute` hingegen handelt es sich direkt um die Klasse aus `de.imotep.core.datamodel`.

Ein Objekt der Klasse `MStateMachine` besteht nach diesem Modell aus einer `RootRegion` und darüber hinaus aus einer Menge von Objekten der Klassen `MAction`, `MEvent`, `MGuard` und `MAttribute`. Des Weiteren kann sie noch Referenzen auf weitere, tiefer verankerte Objekte der Klasse `MRegion` aufweisen. Ein Objekt der Klasse `MRegion` kann aus mehreren Objekten der Klassen `MState`, `MTransition` und `MStateGroup` bestehen, während ein `MState`-Objekt wiederum Objekte der Klasse `MRegion` beinhalten kann. Ein `MState`-Objekt kann eine beliebige Anzahl an eingehenden und ausgehenden Transitionen der Klasse `MTransition` haben. Ein `MTransition`-Objekt hingegen muss auf jeden Fall ein `MState`-Objekt als Ausgangs- und eins als Zielzustand haben. Einem `MTransition`-Objekt können außerdem `MAction`-, `MEvent`- und `MGuard`-Objekte zugeordnet sein, während einem `MState`-Objekt `MAction`-Objekte angeheftet sein können. Zusätzlich kann ein Objekt der Klasse `MState` Referenzen auf ein `MStateGroup`-Objekt besitzen und umgekehrt. Objekte

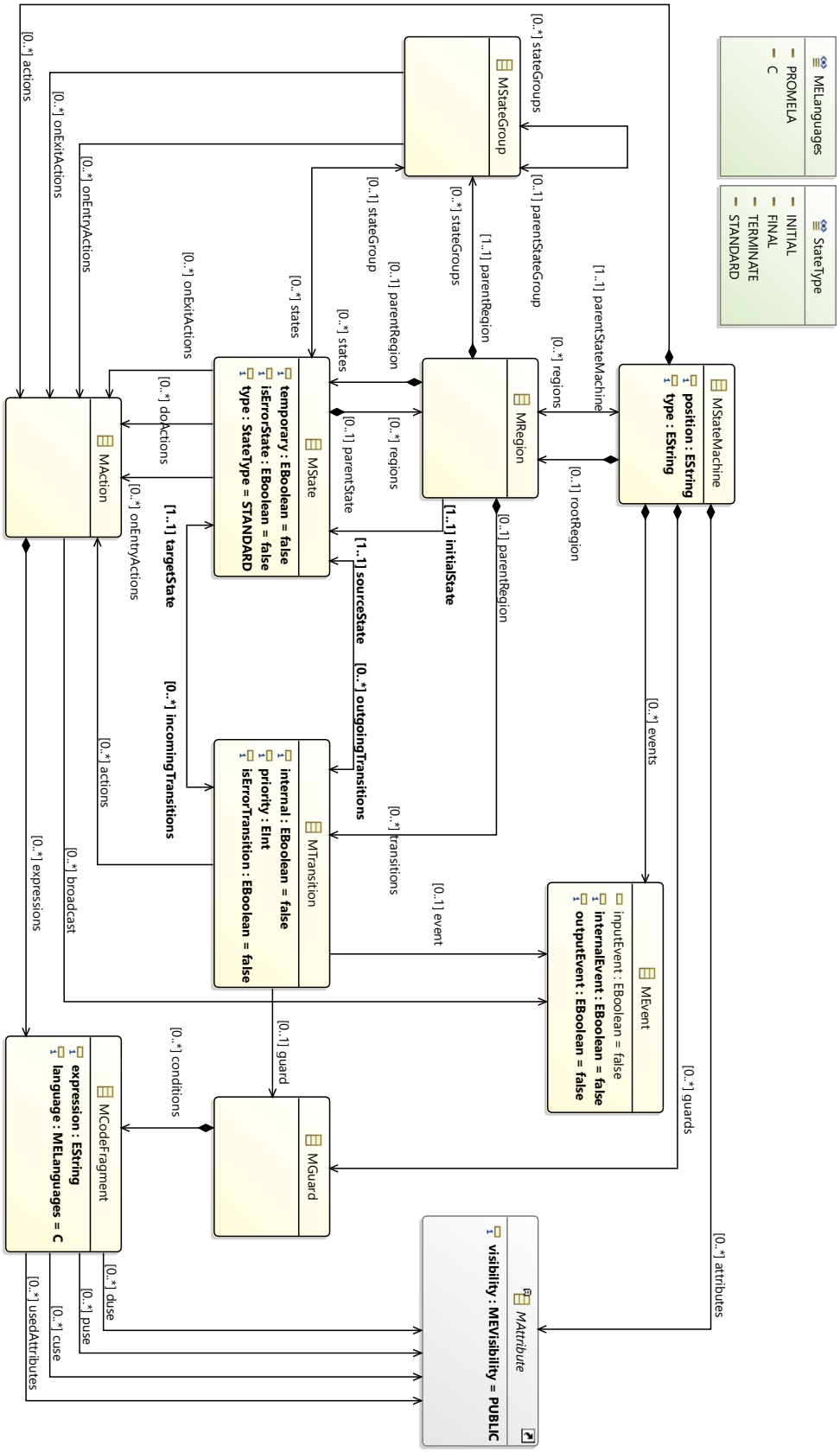


Abbildung 5.4.: Meta-Modell Plug-in `de.imotep.core.behavior`

der Klassen `MAction` und `MGuard` können verschiedene Bedingungen durch Objekte der Klasse `MCodeFragment` enthalten. Letztere beziehen sich auf eine Menge von Attributen `MAttribute`.

de.imotep.variability.annotatedBehavior Dieses Plug-in beschreibt die Struktur eines 150%-Modells und baut dabei wiederum auf dem soeben beschriebenen Plug-in `de.imotep.core.behavior` auf. Dazu wird die Klasse `MAnnotatedEntity` von der Klasse `MBehaviorEntity` des Meta-Modells der normalen State Machine abgeleitet. Ein Objekt der Klasse `MAnnotatedEntity` enthält somit zusätzlich zu Namen und ID eine Annotation in Form eines Strings, der eine aussagenlogische Formel über Feature-Parameter enthält. Von der Klasse `MAnnotatedEntity` erben die Klassen `MAnnotatedStateMachine`, `MAnnotatedRegion`, `MAnnotatedStateGroup`, `MAnnotatedState`, `MAnnotatedTransition` und `MAnnotatedCodeFragment`, sodass diese somit alle ebenfalls eine Annotation besitzen. Diese Klassen implementieren außerdem zusätzlich die Eigenschaften ihrer nicht-annotierten Geschwister-Klassen aus dem Plug-in `de.imotep.core.behavior`. Das Meta-Modell für das Plug-in `de.imotep.variability.annotatedBehavior` ist in Abbildung 5.5 repräsentiert.

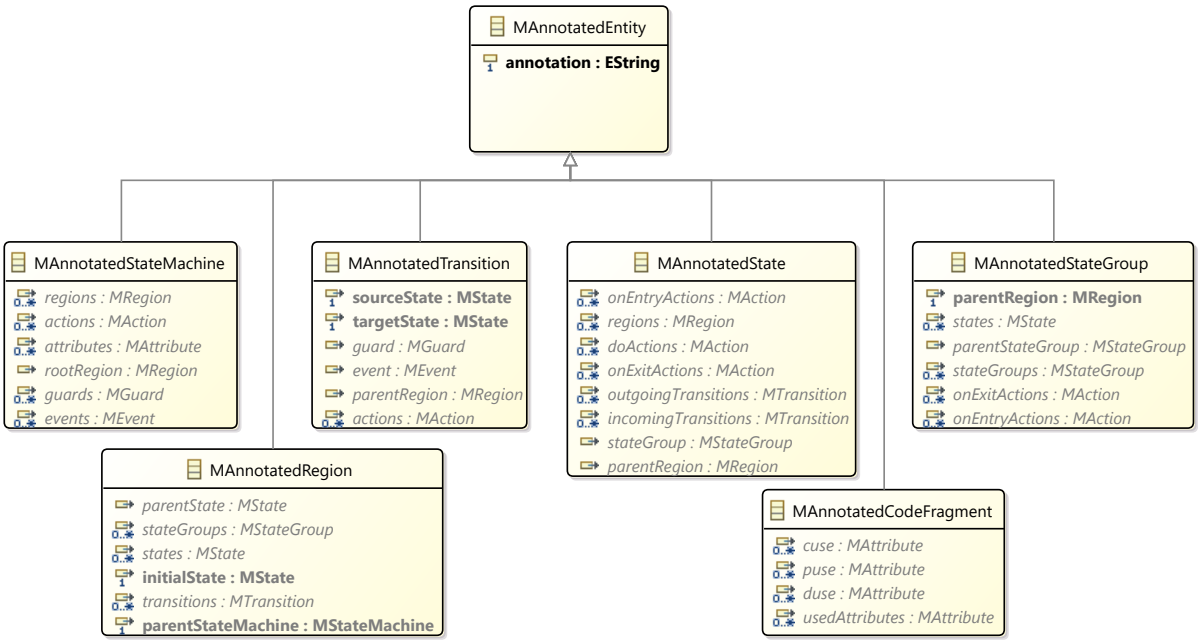


Abbildung 5.5.: Meta-Modell Plug-in `de.imotep.variability.annotatedBehavior`

de.imotep.slicing Das Plug-in `de.imotep.slicing` bietet eine Funktion zum Slicing von normalen State Machines und 150%-State Machines. Das Meta-Modell dieses Plug-ins ist in Abbildung 5.6 dargestellt. Eine ausführliche Beschreibung der ursprünglichen Implementierung ist bei Kamischke [20] zu finden. An dieser Stelle sollen nur kurz die nötigsten Strukturen zur Beschreibung der Erweiterung vorgestellt werden.

Die zentrale Klasse der Slicing-Funktion ist der `MSlicingManager`, der die Funktionen zum Erstellen eines Slices verwaltet und ausführt. Dazu enthält er eine Slicing-Strategie in Form eines Objektes der Klasse `MSlicingStrategy`, welche entweder vorwärts (`MSlicingStrategyForward`) oder rückwärts (`MSlicingStrategyBackward`) sein kann. Zusätzlich beinhaltet er ein Slicing-Kriterium `MSliceCriterium` oder ein davon abgeleitetes 150%-Slicing-Kriterium der Klasse `MSliceCriteri-`

um150. Das MSliceCriterium enthält dabei lediglich ein Objekt der Klasse MTransitionGoal oder MStateGoal, welches entweder auf einen Zustand oder eine Transition verweist und somit das Element für das Kriterium festlegt. Ein Objekt der Klasse MSliceCriterium150 spezifiziert darüber hinaus noch eine aussagenlogische Formel über Feature-Parameter in Form eines Strings. Jedem Kriterium ist ein Objekt der Klasse MSlice zugeordnet, das sich auf ein bestimmtes MStateMachine-Objekt bezieht. Ein MSlice-Objekt verweist auf alle Objekte der Klassen MState und MTransition, die im Slice für dieses Kriterium in Kombination mit dieser State Machine enthalten sind. Die Objekte werden durch die Methode createSlice ausgehend vom MSliceGoal-Objekt rekursiv zum Slice hinzugefügt, wenn eine Abhängigkeit zu einem MState- oder MTransition-Objekt besteht, das sich bereits im Slice befindet.

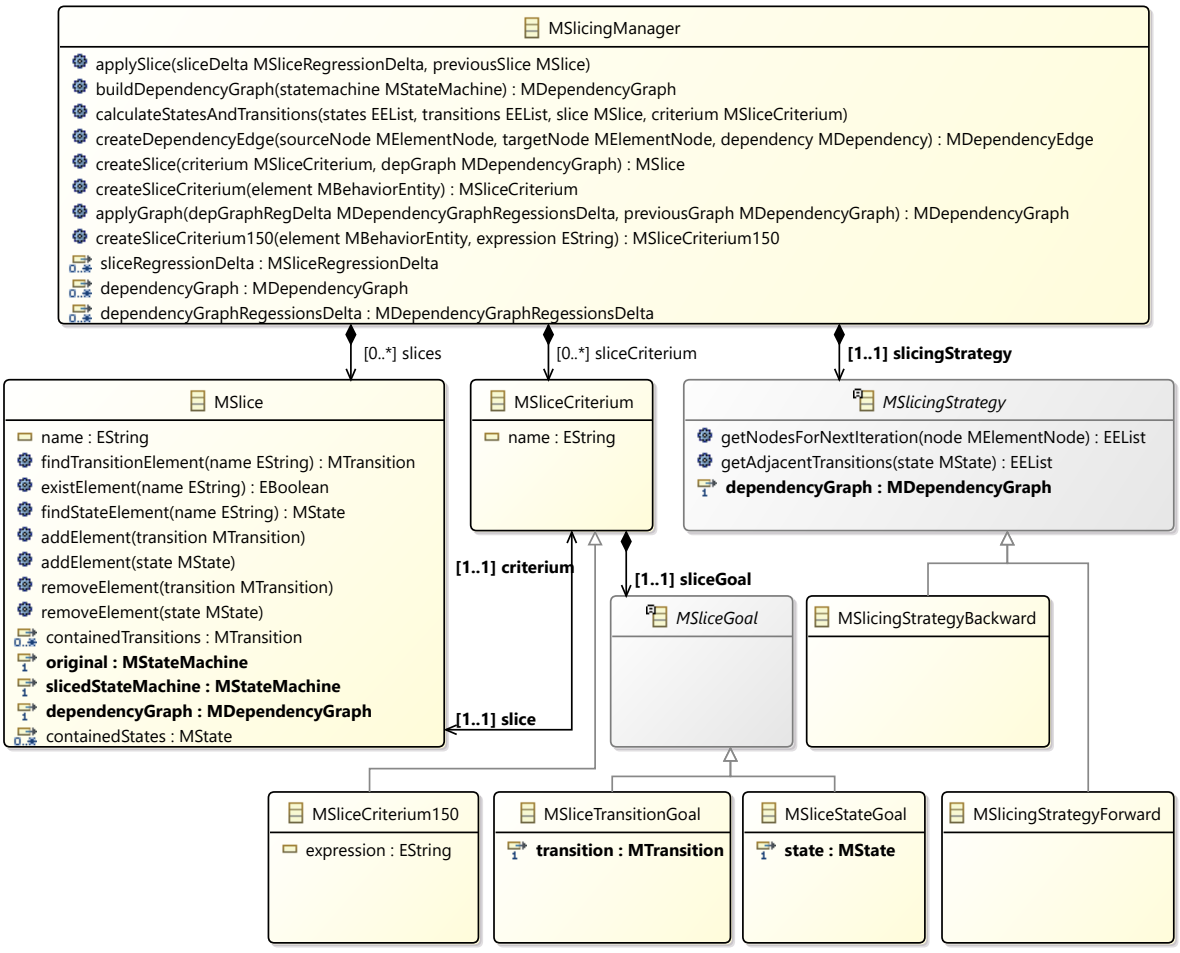


Abbildung 5.6.: Meta-Modell Plug-in de.imotep.slicing

de.imotep.slicing.dependency Die Menge der Objekte in einem Slice wird also durch die Abhängigkeiten zwischen Objekten bestimmt. Welche Abhängigkeiten zwischen den Objekten eines Modells bestehen, wird in einem Abhängigkeitsgraphen festgehalten. Für die Erstellung und Verwaltung eines Abhängigkeitsgraphen für eine State Machine ist das Plug-in de.imotep.slicing.dependency zuständig. Das Meta-Modell dieses Plug-ins ist in Abbildung 5.7 abgebildet.

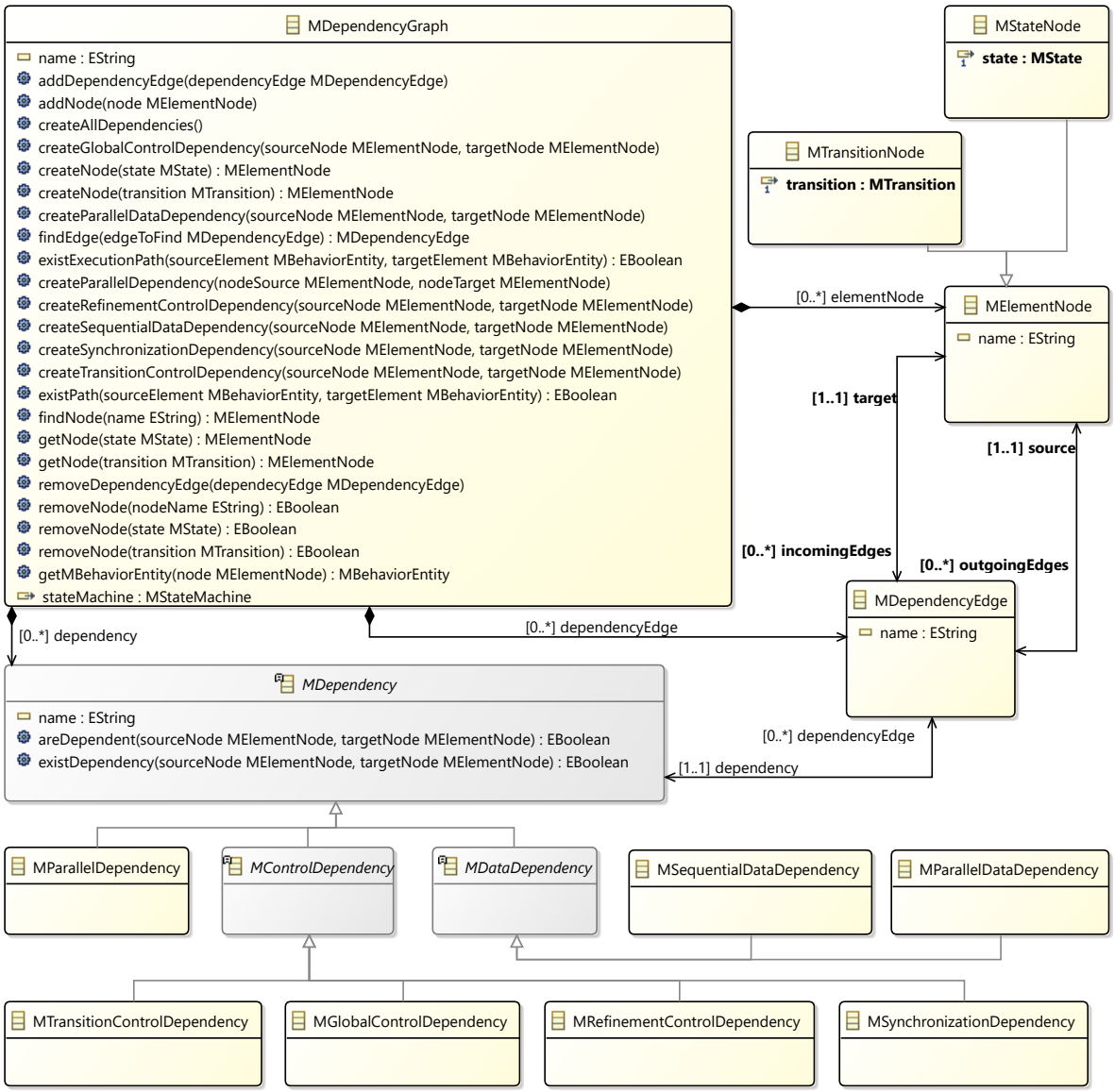


Abbildung 5.7.: Meta-Modell Plug-in `de.imotep.slicing.dependency`

Ein Objekt der Klasse `MDependencyGraph` besteht aus einer Menge von `MElementNode`-Objekten. Ein `MElementNode` kann dabei entweder ein `MStateNode` sein, wenn es sich um einen Zustand einer State Machine handelt, oder ein `MTransitionNode`, falls es eine Transition der State Machine ist. Einem `MElementNode`-Objekt können beliebig viele Objekte der Klasse `MDependencyEdge` zugeordnet sein. Ein `MDependencyEdge`-Objekt verbindet somit zwei `MElementNode`-Objekte, wenn zwischen diesen beiden eine Abhängigkeit besteht. Die Art der Abhängigkeit wird durch die abstrakte Klasse `MDependency` beschrieben. Von dieser lassen sich die verschiedenen Arten der Abhängigkeit, welche in Kapitel 4 vorgestellt wurden, als Unterklassen ableiten. Ein solches Objekt, das die Art der Abhängigkeit spezifiziert, ist jedem `MDependencyEdge`-Objekt zugeordnet. Durch das Setzen eines `MDependencyEdge`-Objekts als ein- oder ausgehende Kante eines `MElementNode`-Objekts wird außerdem definiert, in welche Richtung die Abhängigkeit zwischen zwei `MElementNode`-Objekten besteht.

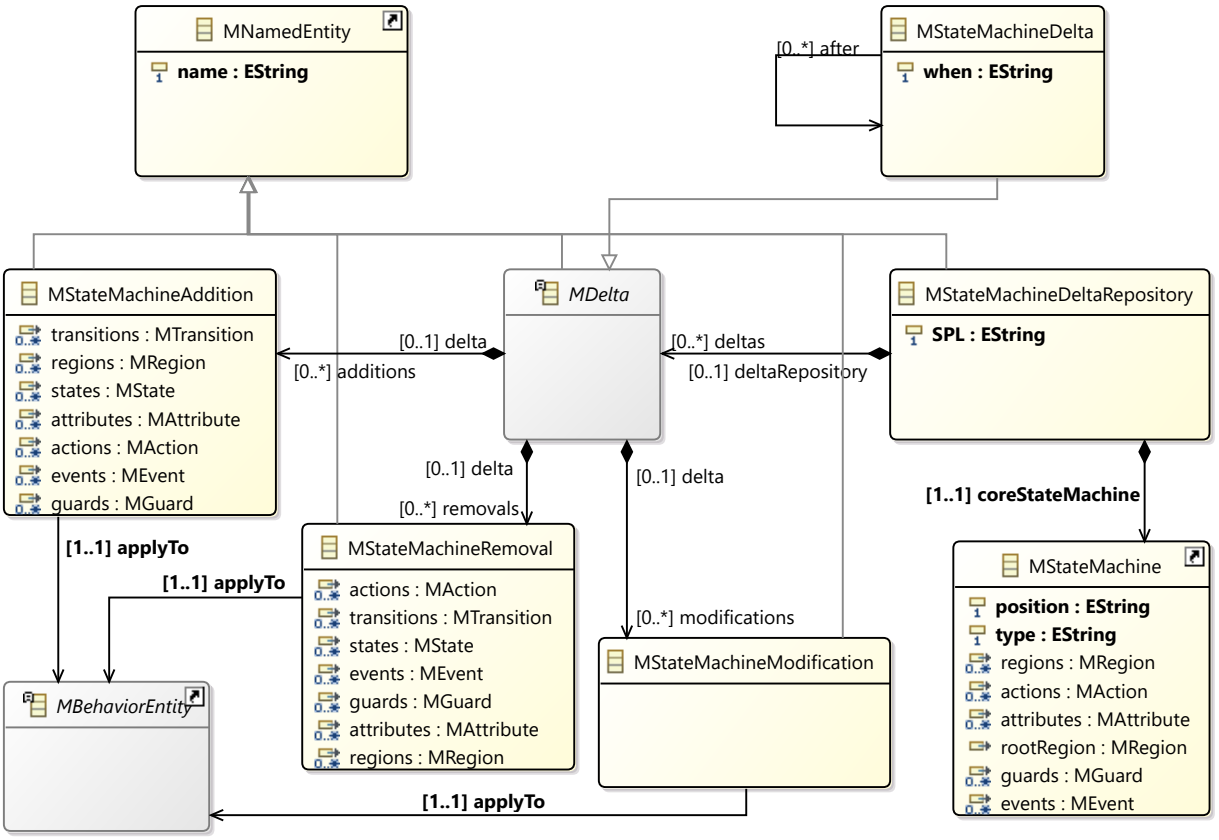


Abbildung 5.8.: Meta-Modell Plug-in `de.imotep.variability.deltaBehavior`

de.imotep.variability.deltaBehavior Abbildung 5.8 zeigt das Meta-Modell des Plug-ins `de.imotep.variability.deltaBehavior`. Ein Delta-Modell ist durch die Klasse `MStateMachineDeltaRepository` beschrieben, welche aus einer `MStateMachine` des Plug-ins `de.imotep.core.behavior` als Kernmodell und einer Menge von Deltas der Klasse `MStateMachineDelta` besteht. Ein `MStateMachineDelta` besitzt durch das Attribut `when` eine Anwendungsbedingung in Form einer aussagenlogischen Formel über Feature-Parameter. Jedes `MStateMachineDelta` setzt sich aus einer beliebigen Menge an Objekten der Klassen `MStateMachineAddition`, `MStateMachineRemoval` und `MStateMachineModification` zusammen. Diese drei Klassen repräsentieren die Element-Operation innerhalb eines Deltas und referenzieren jeweils die Objekte, auf die sie angewendet werden.

de.imotep.dope Die Struktur eines Higher-Order Delta-Modells wird durch das Plug-in `de.imotep.dope` definiert. Das Meta-Modell eines Higher-Order Delta-Modells ist in Abbildung 5.9 abgebildet. Ein Higher-Order Delta-Modell ist dabei ein Objekt der Klasse `HigherOrderDeltaRepository`. Dieses besteht ebenfalls aus einer `MStateMachine` des Plug-ins `de.imotep.core.behavior` als Kernmodell und darüber hinaus aus einer Menge von `HigherOrderDelta`-Objekten. Das Kernmodell ist damit kein komplettes Delta-Modell in Form eines `MStateMachineDeltaRepository`-Objektes, sondern nur eine einfache State Machine. Das erste Delta-Modell wird daher durch die State Machine und das erste Higher-Order Delta gemeinsam repräsentiert. Ein `HigherOrderDelta`-Objekt setzt sich aus beliebig vielen Objekten der Klassen `HigherOrderAdd`, `HigherOrderRemove` und `HigherOrderModify` zusammen, welche die Delta-Operationen eines Higher-Order Deltas ver-

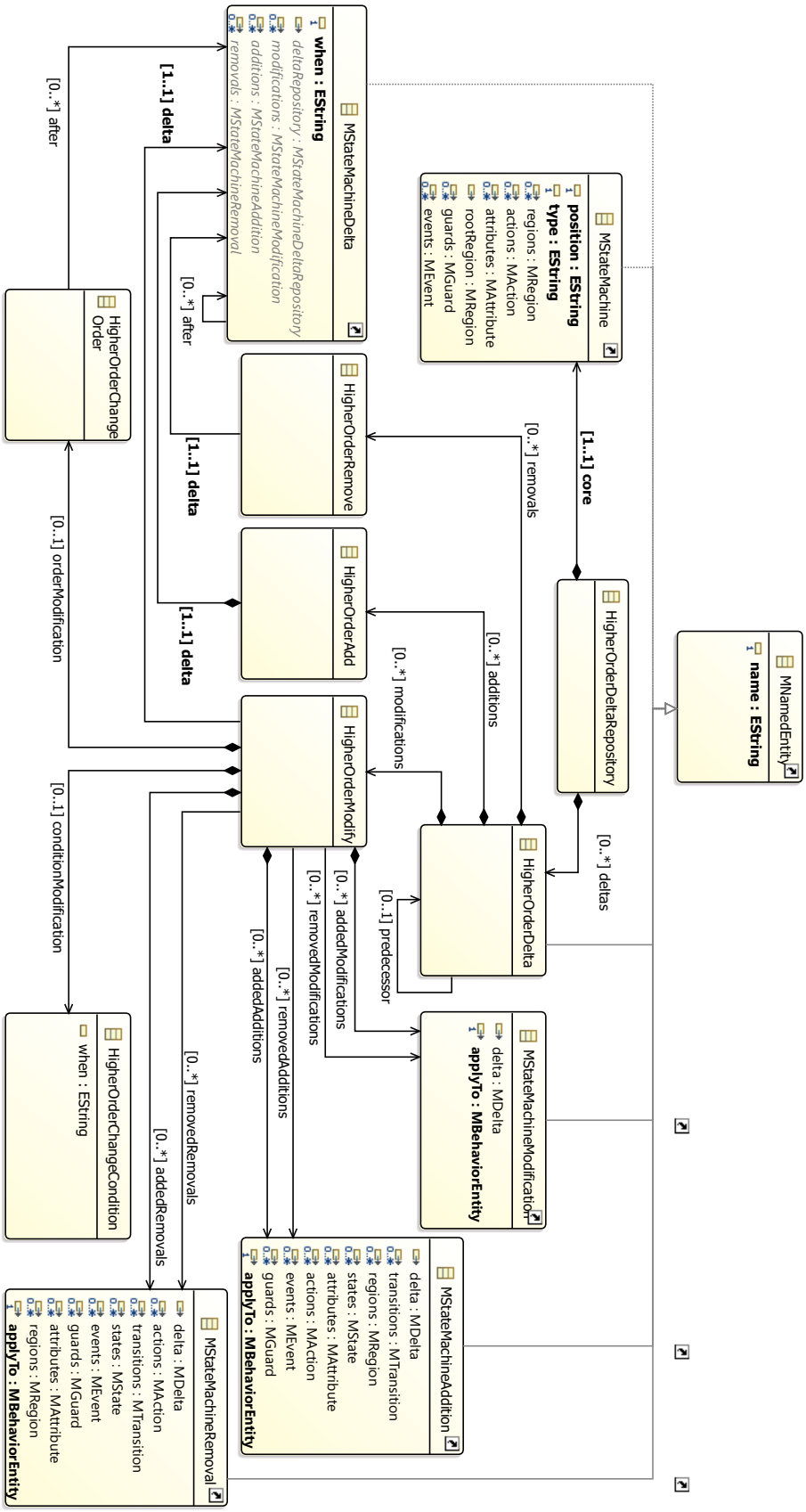


Abbildung 5.9.: Meta-Modell Plug-in `de.imotep.dope`

körpern. `HigherOrderAdd` und `HigherOrderRemove` enthalten ein Delta in Form eines `MStateMachineDelta`-Objektes. Ein `HigherOrderModify` verweist hingegen nur auf ein `MStateMachineDelta` und enthält zusätzlich Objekte der `de.imotep.variability.deltaBehavior`-Klassen `MStateMachineAddition`, `MStateMachineRemoval` und `MStateMachineModification`, die durch die Modifikation zum referenzierten Delta hinzugefügt werden. Außerdem referenziert das `HigherOrderModify` auf weitere Objekte der genannten Klassen, wenn diese aus dem Delta entfernt werden sollen. Darüber hinaus kann es `HigherOrderChangeCondition`- oder `HigherOrderChangeOrder`-Objekte enthalten, um die Anwendungsbedingung oder die Reihenfolge eines `MStateMachineDelta`-Objekts zu verändern.

Nachdem nun die Plug-ins beschrieben wurden, auf denen die umzusetzende Implementierung aufbaut, werden im folgenden Kapitel die neu erstellten beziehungsweise erweiterten Plug-ins vorgestellt.

5.2.2. Datenmodelle der neuen Implementierung

Im Einklang mit den bereits bestehenden Plug-ins wurden die Datenmodelle der neuen Plug-ins mit Hilfe des Eclipse Modeling Frameworks als EMF Meta-Modelle definiert. Auf Basis dieser Meta-Modelle ist das Modeling Framework in der Lage, die Java-Klassen der Modellelemente automatisch zu generieren. Für die 175%-Modelle kann das Framework außerdem verwendet werden, um Adapterklassen zur Darstellung und Bearbeitung zu erzeugen, als auch einen Editor für 175%-Modelle anzufertigen.

Die folgenden Plug-ins werden zur Umsetzung der Implementierung von 175%-Modellen und Transformationsalgorithmus neu erstellt:

- **de.imotep.evolution.temporalAnnotation:** Die Java-Klassen der Elemente einer 175%-State Machine
- **de.imotep.evolution.temporalAnnotation.edit:** Die Adapterklassen zur Bearbeitung der 175%-State Machine
- **de.imotep.evolution.temporalAnnotation.editor:** Der Editor für 175%-State Machines
- **de.imotep.evolution.transformation:** Die Java-Klassen, die zum Durchführen einer Transformation von Higher-Order Delta-Modellen in 175%-Modelle benötigt werden

Die Implementierung des Slicings von 175%-State Machines erfolgt hingegen im folgenden, bereits bestehenden Plug-in:

- **de.imotep.slicing:** Funktioniert nun zusätzlich für 175%-Modelle

de.imotep.evolution.temporalAnnotation Die Struktur für 175%-State Machines wurde im Plug-in `de.imotep.evolution.temporalAnnotation` definiert. Sie baut genauso wie die Struktur der 150%-State Machines aus dem Plug-in `de.imotep.variability.annotatedBehavior` auf den beiden Plug-ins `de.imotep.core.datamodel` und `de.imotep.core.behavior auf`.

Abbildung 5.10 zeigt das Meta-Modell eines 175%-Modells, welches einen ähnlichen Aufbau wie das Meta-Modell der 150%-State Machines in Abbildung 5.5 aufweist. Allerdings wurde hierbei die

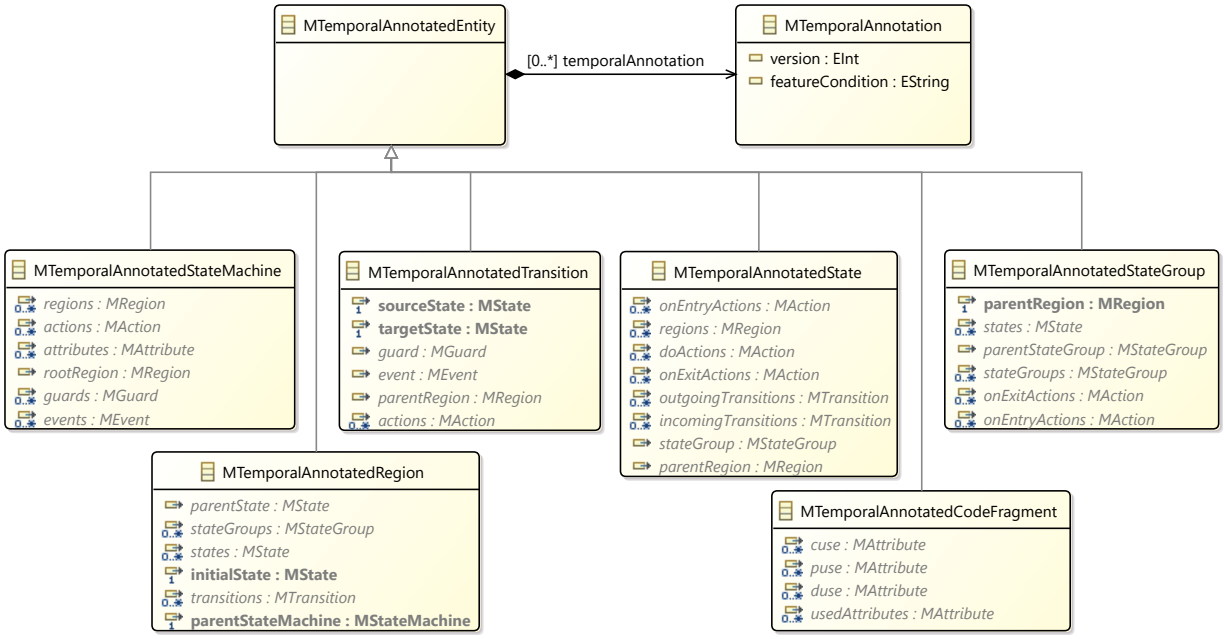


Abbildung 5.10.: Meta-Modell Plug-in `de.imotep.evolution.temporalAnnotation`

Klasse `MAnnotatedEntity` durch die Klasse `MTemporalAnnotatedEntity` ausgetauscht. Die gleichen Klassen, die im Plug-in `de.imotep.variability.annotatedBehavior` von `MAnnotatedEntity` geerbt haben, erben hier von `MTemporalAnnotatedEntity`. Ein Objekt der Klasse `MTemporalAnnotatedEntity` enthält eine Menge von Objekten der neuen Klasse `MTemporalAnnotation`. Ein `MTemporalAnnotation`-Objekt besitzt die Eigenschaften `version` in Form eines Integers und `featureCondition` in Form eines Strings. Ein solches Objekt repräsentiert folglich genau ein Tupel aus Version und Feature-Bedingung (s. Kapitel 3.2). Die gesamte Menge an einem Element zugeordneten `MTemporalAnnotation`-Objekten ergibt somit die vollständige Annotation dieses Elements.

Die Struktur der 150%-Modelle kann an dieser Stelle nicht wiederverwendet werden, da jeweils ein Tupel bestehend aus Version und Feature-Bedingung als eigenes Objekt dargestellt werden soll, damit eine eindeutige Zuordnung zwischen diesen Werten stattfinden kann. Ein Objekt der Klasse `MAnnotatedEntity` enthält bereits eine einzelne Feature-Bedingung als Annotation und kann daher an dieser Stelle nicht als Superklasse für die Klasse `MTemporalAnnotatedEntity` dienen.

de.imotep.evolution.transformation Dieses Plug-in realisiert die Funktion zur Transformation von Higher-Order Delta-Modellen in 175%-Modelle. Das Meta-Modell dieses Plug-ins ist in Abbildung 5.11 dargestellt. Die wesentliche Klasse zur Umsetzung dieser Funktion ist der `TransformationManager`, welcher eine Referenz auf das eingegebene `HigherOrderDeltaRepository` und auf die zu erzeugende `MTemporalAnnotatedStateMachine` besitzt. Der Manager enthält zum einen außerdem eine Menge von Objekten der Klasse `AnnotationProperties`. Diese referenzieren jeweils auf ein Element des zu erzeugenden 175%-Modells und speichern die Delta-Anwendungsbedingung für das letzte, betrachtete Delta, das eine Add-Operation auf diesem Element ausführt. Des Weiteren enthält ein `AnnotationProperties`-Objekt einen Boolean-Wert mit der Information, ob das Element zurzeit ein offenes Intervall besitzt und somit für zukünftige Versionen weiterhin gültig ist. Zum

anderen besitzt der `TransformationManager` ein Objekt der Klasse `UpdateList`. Ein solches Objekt setzt sich wiederum aus mehreren Objekten der Klasse `UpdateInformation` zusammen. Ein `UpdateInformation`-Objekt besitzt eine Referenz auf ein `HigherOrderModify`-Objekt und speichert eine Version in Form eines Integers. Diese Objekte werden gebraucht, um die Deltas, deren aktuelle Zusammensetzung in der Liste `deltaList` gespeichert wird, für jede Version zu aktualisieren. Darüber hinaus enthält der Manager ein Objekt der Klasse `SortedHODList`, welches wiederum aus mehreren `SortedHOD`-Objekten bestehen kann. Ein `SortedHOD`-Objekt besteht aus den Kombinationen aus Delta- und Element-Operationen, die in Form von `Combination`-Objekten für genau ein Higher-Order Delta nach Priorität geordnet wurden (s. Kapitel 4.2). Zusätzlich speichert das `SortedHOD` die Version, die das Higher-Order Delta repräsentiert, als Integer.

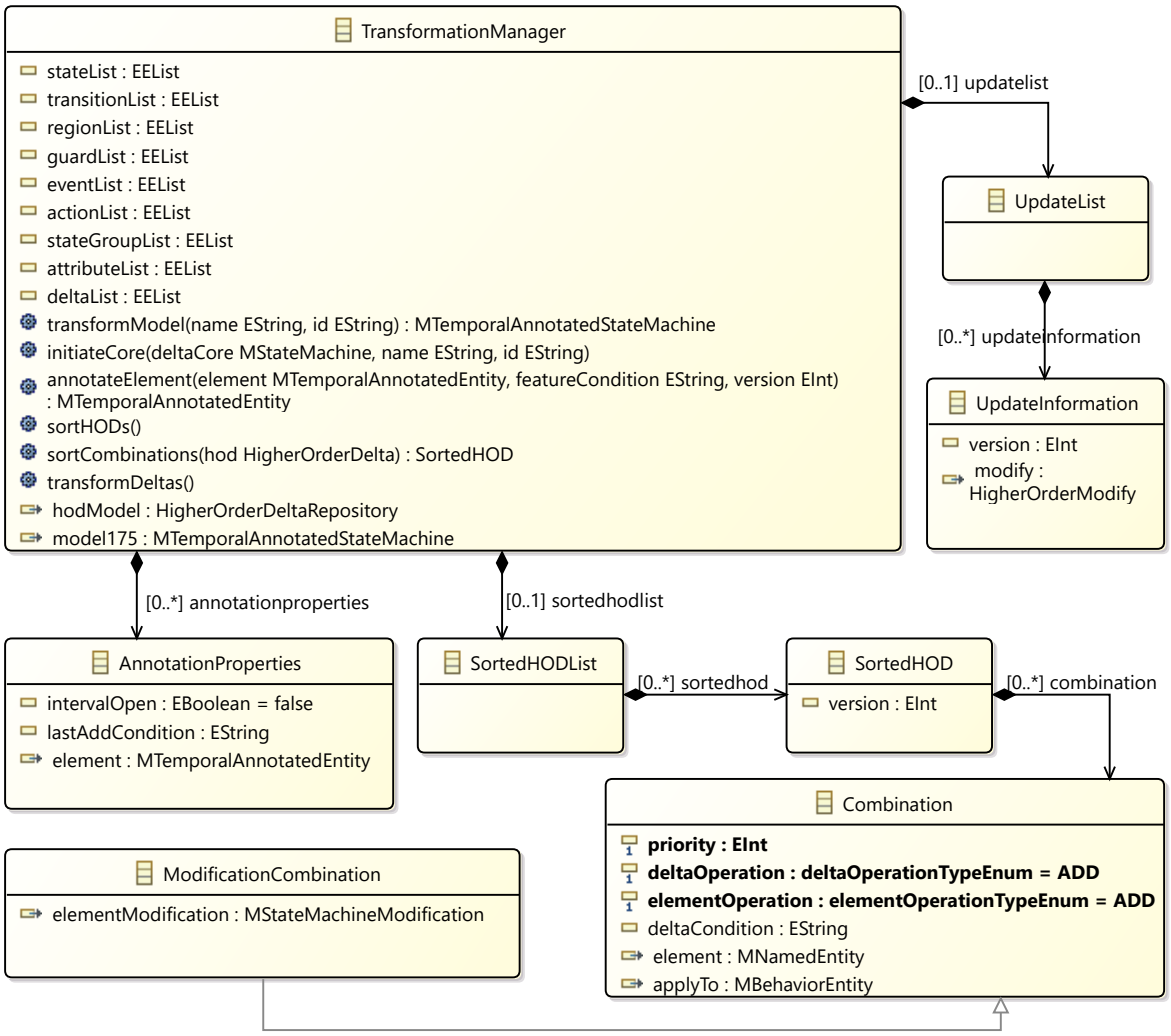


Abbildung 5.11.: Meta-Modell Plug-in `de.imotep.evolution.transformation`

Ein `Combination`-Objekt speichert sowohl eine Priorität (`priority`) als auch den Typ der Delta-Operation und der Element-Operation. Darüber hinaus enthält es die Anwendungsbedingung des Deltas, von dem die Operationen stammen, als String und Referenzen auf das Element, auf dem die Operationen ausgeführt werden, als auch das Element, in welchem das erste Element enthalten

ist (`applyTo`). Diese Menge an Informationen ist nötig, da der Manager beim Vergeben der Annotation nur noch mit den geordneten Kombinationen arbeitet und nicht mehr mit den originalen `HigherOrderDelta`-Objekten. Ferner besitzt der `TransformationManager` eine Menge von Listen, in denen die `AnnotationProperties`-Objekte entsprechend der referenzierten Objekte klassenspezifisch eingeordnet sind. Auf diese Weise müssen nicht alle existierenden Objekte durchsucht werden, um ein bestimmtes Objekt zu finden, sondern lediglich die Objekte, welche nur auf Objekte der gesuchten Klasse referenzieren. Der Manager spezifiziert außerdem die Methoden zur Durchführung des Algorithmus, welche in Kapitel 5.4 näher erläutert werden.

de.imotep.slicing Die Implementierung des 175%-Slicings wird direkt in dem bestehenden Plug-in `de.imotep.slicing` umgesetzt. Dazu wird die bisherige Implementierung an ausgewählten Stellen verändert oder erweitert. Der veränderte Teil des Slicing-Meta-Modells ist in Abbildung 5.12 visualisiert. Dort wird die neue Klasse `MSliceCriterium175` hinzugefügt. Diese wird von der Klasse des 150%-Kriteriums abgeleitet und enthält zusätzlich eine Liste von Versionen in Form von Integern. Neben dieser Erweiterung erhält außerdem die Klasse `MSlicingManager` eine neue Methode zur Erstellung eines Objektes der Klasse `MSliceCriterium175`.

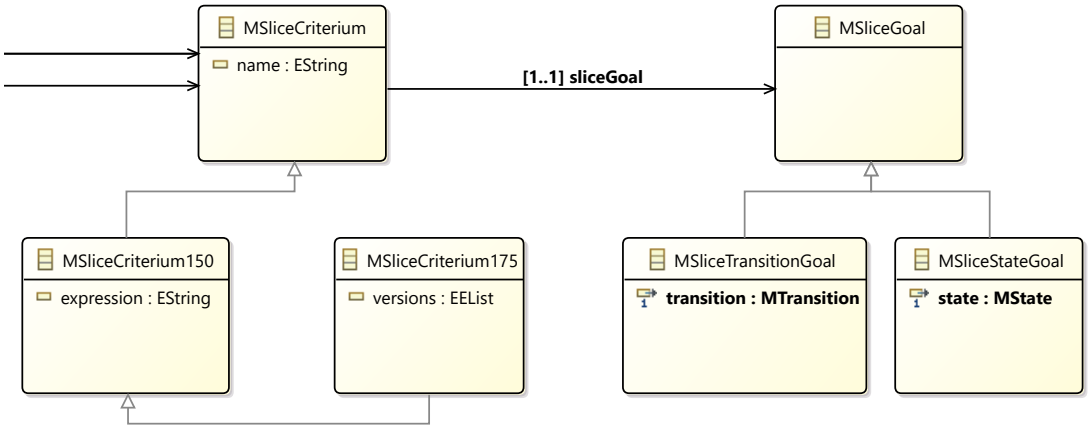


Abbildung 5.12.: Ausschnitt des erweiterten Slicing-Meta-Modells

Im folgenden Kapitel werden die Abhängigkeiten und das Zusammenspiel zwischen den verschiedenen Plug-ins beschrieben.

5.2.3. Plug-in Architektur

Zur Ermöglichung einer leichteren Erweiterbarkeit wird das Eclipse Plug-in zur Umsetzung der gewünschten Anforderungen in Form von mehreren kleinen, separaten Plug-ins organisiert. Auf diese Weise wird sowohl eine leichtere Verwendung einzelner Plug-ins in anderen Projekten ermöglicht, die nur Teile der Implementierung wie beispielweise die 175%-Modelle benötigen, als auch eine unkompliziertere Erweiterung durch weitere, zusätzliche Plug-ins erlaubt. Ebenfalls können einzelne Plug-ins erweitert werden, wie es bereits hier für das `de.imotep.slicing`-Plug-in der Fall ist.

In Abbildung 5.13 ist das Zusammenspiel der bestehenden und neu erstellten Plug-ins abgebildet, wobei die Pfeile Abhängigkeiten zwischen den verschiedenen Plug-ins beschreiben. Dabei ist das Ausgangs-Plug-in auf das Ziel-Plug-ins angewiesen. Dies resultiert daher, dass das Ausgangs-Plug-in Funktionalitäten verwendet, die durch das Ziel-Plug-in implementiert werden. Entsprechend

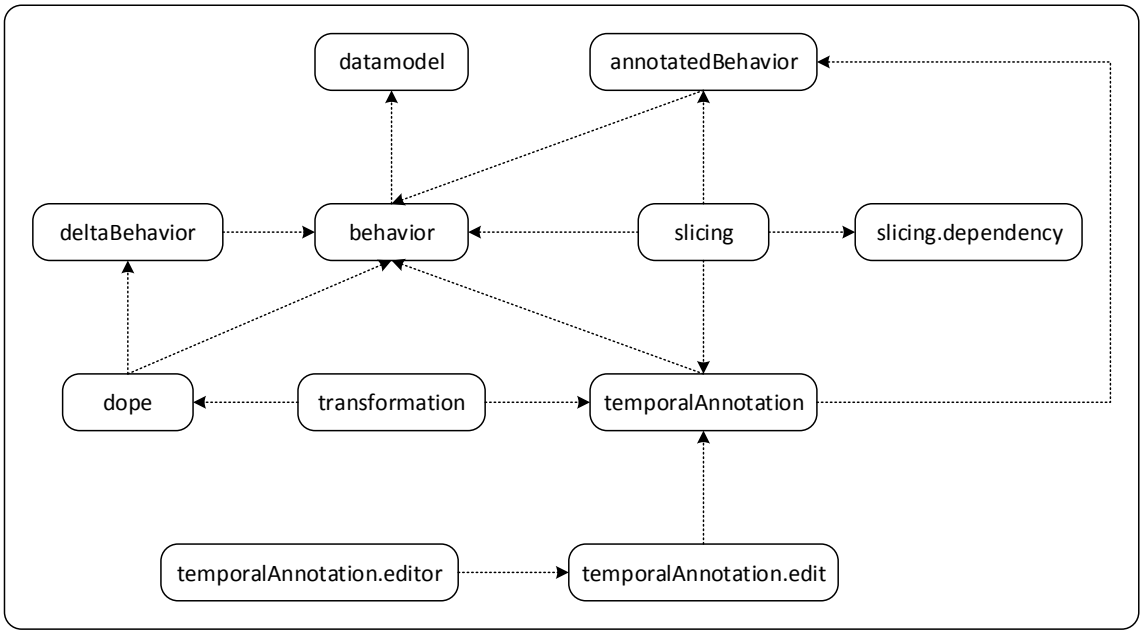


Abbildung 5.13.: Architektur des Plug-ins

Abbildung 5.13 ist das neue Plug-in `de.imotep.evolution.transformation` folglich abhängig vom bestehenden Plug-in `de.imotep.dope`, da die Transformation ein `HigherOrderDeltaRepository` als Eingabe-Modell verarbeitet, und vom neuen Plug-in `de.imotep.evolution.temporalAnnotation`, da eine `MTemporalAnnotatedStateMachine` als Ausgabe-Modell geliefert wird. Die Plug-ins `de.imotep.evolution.temporalAnnotation` und `de.imotep.dope` sind wiederum beide vom Plug-in `de.imotep.core.behavior` abhängig, da `de.imotep.dope` eine `MStateMachine` als Kern des `HigherOrderDeltaRepository` verwendet, während die Klassen von `de.imotep.evolution.temporalAnnotation` von den Klassen von `de.imotep.core.behavior` abgeleitet sind. Zusätzlich ist `de.imotep.dope` vom Plug-in `de.imotep.variability.deltaBehavior` abhängig, da die Higher-Order Deltas, die dort definierten Deltas enthalten. Da ein Delta-Modell ebenfalls eine State Machine als Kernmodell enthält, ist `de.imotep.variability.deltaBehavior` ebenfalls von `de.imotep.core.behavior` abhängig. Letzteres ist im Gegenzug abhängig vom Plug-in `de.imotep.core.datamodel`, da alle Klassen der State Machine von dessen Klassen abgeleitet sind.

Das Plug-in `de.imotep.slicing` ist sowohl von den Plug-ins `de.imotep.evolution.temporalAnnotation`, `de.imotep.core.behavior` und `de.imotep.variability.annotatedBehavior` abhängig, da es für das Slicing auf 175%-, 150%- oder normalen State Machines operiert, als auch vom Plug-in `de.imotep.slicing.dependency`, da es dieses verwendet, um einen Abhängigkeitsgraphen für das jeweilige Modell zu erstellen. `de.imotep.variability.annotatedBehavior` ist ebenfalls abhängig von `de.imotep.core.behavior`, da die Klassen der 150%-State Machine von den Klassen der normalen State Machine abgeleitet sind. Darüber hinaus ist das Plug-in `de.imotep.evolution.temporalAnnotation.editor` von `de.imotep.evolution.temporalAnnotation.edit` abhängig, welches im Gegenzug vom Plug-in `de.imotep.evolution.temporalAnnotation` abhängig ist. Das Editor-Plug-in ist dabei vom Edit-Plug-in abhängig, da dessen Adapterklassen zur korrek-

ten Anzeige und Bearbeitung der Elemente der 175%-State Machine im Editor benötigt werden. Die Adapterklassen wiederum basieren auf den Klassen der 175%-State Machine aus `de.imotep.evolution.temporalAnnotation`. Zusätzlich sind sämtliche Plug-ins abhängig vom Eclipse Modeling Framework, welches von Eclipse als Plug-in angeboten wird, da sie auf diesem basieren. Daher muss dieses ebenfalls zur Verfügung stehen, damit die vorgestellten Plug-ins funktionieren.

Im Folgenden wird nun die Realisierung der Funktionalitäten durch Hervorhebung einzelner Aspekte näher erläutert.

5.3. 175%-Modelle und Versionsableitung

Das Datenmodell einer 175%-State Machine wurde bereits in Kapitel 5.2.2 beschrieben. Durch die Plug-ins `de.imotep.evolution.temporalAnnotation.edit` und `de.imotep.evolution.temporalAnnotation.editor` besteht außerdem die Möglichkeit, ein 175%-Modell per Editor zu erstellen. Dazu kann ein neues Projekt und über *New* > *Other...* ein neues Modell erstellt werden.

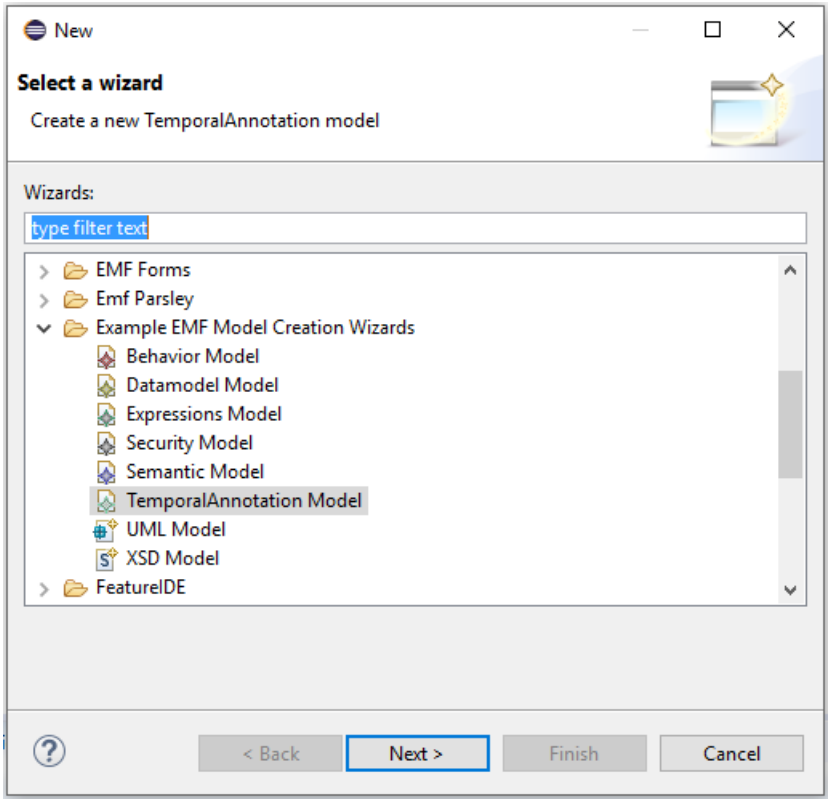


Abbildung 5.14.: Wizard zur Erstellung eines 175%-Modells

Der Wizard zur Erstellung eines 175%-Modell ist, wie in Abbildung 5.14 dargestellt, im Untermenü *Example EMF Model Creation Wizards* zu finden. Wird dieser ausgewählt, muss im nächsten Schritt der Name und der Speicherort bestimmt werden. Danach wird der Benutzer gebeten, ein Modell-Objekt auszuwählen. Für ein vollständiges Modell muss an dieser Stelle das Objekt *MTemporal Annotated State Machine* selektiert werden.

Zusätzlich wird im Plug-in `de.imotep.evolution.temporalAnnotation` eine Möglichkeit implementiert, aus einem existierenden 175%-Modell eine Version als 150%-Modell abzuleiten. Der

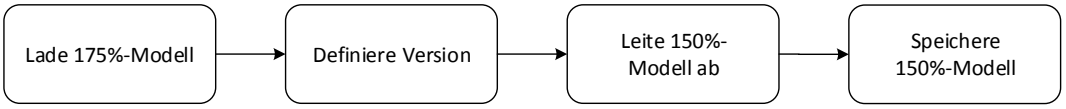


Abbildung 5.15.: Arbeitsablauf Versionsableitung

Arbeitsablauf dieser Funktionalität ist in Abbildung 5.15 abgebildet. Für die Versionsableitung wird die Klasse `VersionDerivation` im Package `de.imotep.evolution.temporalAnnotation` verwendet. In dieser Klasse ist die Methode `deriveVersion(MTemporalAnnotatedStateMachine m175, int version)` enthalten, der ein 175%-State Machine-Objekt und eine Version in Form eines Integers übergeben werden. Die Methode erstellt sodann ein Objekt der Klasse `MAnnotatedStateMachine` aus dem Plug-in `de.imotep.variability.annotatedBehavior`, welches die gleichen Eigenschaften wie das 175%-State Machine-Objekt erhält. Mit Hilfe eines TreeIterators werden dann sämtliche Objekte des 175%-Modells nach dem als RootRegion deklarierten `MTemporalAnnotatedRegion`-Objekt durchsucht. Ausgehend von diesem Objekt werden rekursiv alle enthaltenen Objekte, welche für die eingegebene Version gültig sind, als 150%-Objekte kopiert und zu der 150%-State Machine hinzugefügt. Ein 175%-Objekt ist für die eingegebene Version genau dann gültig, wenn es ein `MTemporalAnnotation`-Objekt enthält, dessen Version mit der eingegebenen Version übereinstimmt. Die Feature-Bedingung aus diesem `MTemporalAnnotation`-Objekt wird dann gleichzeitig als Feature-Bedingung für das 150%-Objekt gesetzt.

```
1  public MAnnotatedStateMachine initiateDerivation(String inputUrl, int version,
2      String outputFolder) {
3
4    MTemporalAnnotatedStateMachine m175 =
5        (MTemporalAnnotatedStateMachine) ModelLoader.load("xmi", inputUrl);
6
7    VersionDerivation vd = new VersionDerivation();
8    MAnnotatedStateMachine m150 = vd.deriveVersion(m175, version);
9
10   String outputUrl = outputFolder + "/150Model_" + m150.getName() + ".xmi";
11   ModelLoader.save(m150, null, "xmi", outputUrl);
12   System.out.println("Model saved as " +outputUrl);
13
14   return m150;
15 }
```

Codefragment 5.1.: Implementierung der Initialisierung der Versionsableitung

Zum Ausführen der Versionsableitung existiert im selben Package die Klasse `RunDerivation`. Diese Klasse enthält die Methode `initiateDerivation(String inputUrl, int version, String outputFolder)`, welche in Codefragment 5.1 abgebildet ist. Diese Methode lädt mit Hilfe der Klasse `ModelLoader`, welche ebenfalls im selben Package zu finden ist, zunächst in Zeile 5 eine 175%-State Machine-Modell-Datei anhand der Speicheradresse des Modells ein. Danach erzeugt die Methode ein `VersionDerivation`-Objekt und ruft die beschriebene Methode `deriveVersion` auf (Zeile 8).

Anschließend wird das Modell mit Hilfe der `ModelLoader`-Klasse im Ordner, dessen Speicheradresse als `outputFolder` übergeben wurde, gespeichert (Zeile 11) und danach außerdem als Rückgabewert zurückgegeben. Diese Methode kann beispielsweise zur Einbettung der Versionsableitung in andere Programme verwendet werden.

Soll das 150%-Modell lediglich zur Weiterverarbeitung abgeleitet, jedoch nicht gespeichert werden, kann die Methode `initiateDerivation(String inputUrl, int version)` verwendet werden. Diese zeigt den gleichen Aufbau wie die Methode in Codefragment 5.1, nur ohne die Zeilen 10 bis 12. Falls das Higher-Order Delta-Modell, das transformiert werden soll, nicht mehr geladen werden muss, sondern durch einen Programmaufruf übergeben werden kann, steht die Methode `initiateDerivation(MTemporalAnnotatedStateMachine m175, int version, String outputFolder)` zur Verfügung. Bei dieser Methode fehlt im Vergleich zur Methode aus 5.1 die Zeile 5. Muss weder das Eingabe-Modell geladen noch das Ausgabe-Modell gespeichert werden, kann die Methode `initiateDerivation(MTemporalAnnotatedStateMachine m175, int version)` benutzt werden. Diese verzichtet im Vergleich zu Codefragment 5.1 auf die Zeilen 5 sowie 10 bis 12. Werden diese Methoden verwendet, muss sowohl für die Speicheradresse des Eingabe-Modells als auch des Ausgabe-Ordners der komplette Pfad angegeben werden.

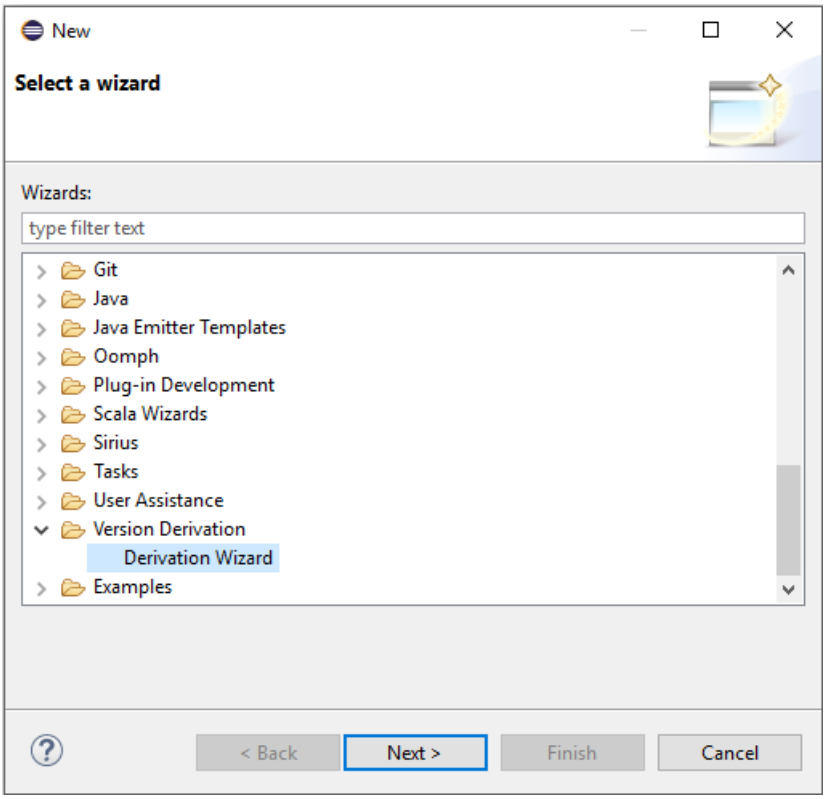


Abbildung 5.16.: Aufrufen eines Wizard zur Versionsableitung

Zur einfachen Anwendung der Versionsableitung ohne Einbettung in ein anderes Programm wurde außerdem ein Wizard als Extension für das Plug-in erstellt. Dieser *Derivation Wizard* kann, wie in Abbildung 5.16 abgebildet, über *New* > *Other...* im Untermenü *Version Derivation* ausgewählt werden. Abbildung 5.17 zeigt den Aufbau des Wizards. Dort muss das 175%-Modell ausgewählt werden, aus welchem eine Version abgeleitet werden soll, sowie die Version bestimmt werden, die

abgeleitet werden soll. Als letztes muss noch der Ordner bestimmt werden, in dem das abgeleitete Modell gespeichert werden soll. Mit einem Klick auf den Finish-Button wird die Methode `derive(String inputUrl, int version, String outputFolder)`, die sich ebenfalls in der Klasse `RunDerivation` befindet, mit den eingegebenen Werten aufgerufen. Diese Methode besitzt den gleichen Aufbau wie die Methode `initiateDerivation(String inputUrl, int version, String outputFolder)` aus Codefragment 5.1, liefert jedoch keinen Rückgabewert.

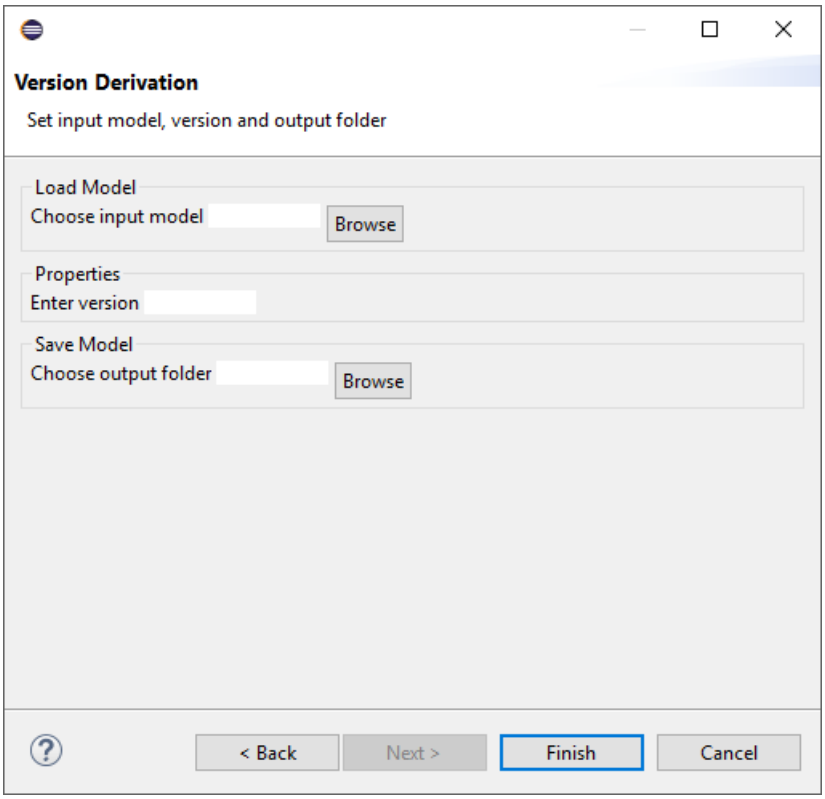


Abbildung 5.17.: Wizard zur Ableitung einer Version aus einem 175%-Modell

Im folgenden Kapitel wird die Implementierung der automatischen Erstellung einer 175%-State Machine mit Hilfe der Modelltransformation beschrieben.

5.4. Modelltransformation

Die Implementierung des Algorithmus zur Transformation von Higher-Order Delta-Modellen in 175%-Modelle erfolgt im neuen Plug-in `de.imotep.evolution.transformation`. Der Arbeitsablauf für diese Funktionalität ist in Abbildung 5.18 dargestellt. Die einzelnen Schritte in dieser Abbildung sind die wesentlichen Phasen, in die sich die Transformation hauptsächlich einteilen lässt.

Wie in Kapitel 5.2.2 erwähnt, ist bei dieser Funktionalität die Klasse `TransformationManager` die Klasse, welche die Transformation steuert. Demnach muss für die Transformation zunächst eine Instanz des Managers erzeugt werden. Nachdem das umzuwandelnde Higher-Order Delta-Modell ebenfalls wie bei der Versionsableitung mit einer Klasse `ModelLoader` geladen wurde, wird das Modell dieser Manager-Instanz mit dem Befehl `setHodModel(HigherOrderDeltaRepository hodModel)` übergeben. Die Transformation wird dann durch die Methode `transformModel(String name, String id)` des Manager gestartet.

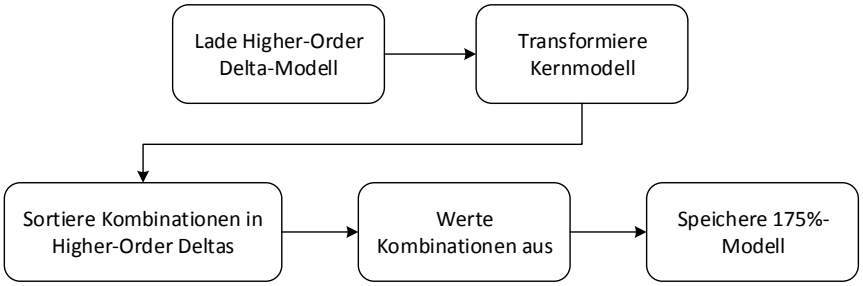


Abbildung 5.18.: Arbeitsablauf Transformation

In der Methode `transformModel(String name, String id)` wird zuerst die Methode `initiateCore(MStateMachine deltaCore, String name, String id)` aufgerufen. Dieser Methode wird das als `MStateMachine`-Objekt enthaltene Kernmodel des Higher-Order Delta-Modells übergeben. Außerdem werden Name und ID für das zu erstellende `MTemporalAnnotatedStateMachine`-Objekt, also die 175%-State Machine, übergeben. Die Methode `initiateCore(MStateMachine deltaCore, String name, String id)` erstellt dann ein `MTemporalAnnotatedStateMachine`-Objekt und setzt Name und ID entsprechend der übergebenen Werte. Danach annotiert der Manager das Objekt mit den Werten `true` für die Feature-Bedingung und `0` als Version. Dies ist die Standard-Annotation, die alle Objekte des Kernmodells zu Beginn erhalten.

```
1  public MTemporalAnnotatedEntity annotateElement(MTemporalAnnotatedEntity element,
2      String featureCondition, int version) {
3
4    boolean annotationExists = false;
5    if (element.getTemporalAnnotation().size() > 0) {
6      MTemporalAnnotation lastAnnotation =
7          element.getTemporalAnnotation().get(element.getTemporalAnnotation().size()-1);
8      if (lastAnnotation.getVersion() == version) {
9        lastAnnotation.setId(lastAnnotation.getId().replace(
10           lastAnnotation.getFeatureCondition(), featureCondition));
11       lastAnnotation.setFeatureCondition(featureCondition);
12       annotationExists = true;
13     }
14   }
15
16   if (annotationExists == false) {
17     AnnotationProperties elementProperties = fetchProperties(element);
18     if (elementProperties.isIntervalOpen() == true) {
19       checkPreviousAnnotations(element, version);
20     }
21     addNewAnnotation(element, featureCondition, version);
22   }
23
24   return element;
25 }
```

Codefragment 5.2.: Implementierung für die Annotierung eines Elements

Annotiert wird ein Objekt mit Hilfe der Methode `annotateElement(MTemporalAnnotatedEntity element, String featureCondition, int version)` aus Codefragment 5.2. Dabei wird zunächst kontrolliert, ob das Element schon ein `MTemporalAnnotation`-Objekt für die übergebene Version besitzt (Zeile 8). Sollte dies der Fall sein, wird kein neues `MTemporalAnnotation`-Objekt erstellt, sondern die Feature-Bedingung des alten `MTemporalAnnotation`-Objekts wird durch die neue Feature-Bedingung ersetzt (Zeile 11). Besitzt das Element noch keine Annotation für diese Version, wird das dem Element zugeordnete `AnnotationProperties`-Objekt gesucht und überprüft, ob das Element ein offenes Intervall aufweist (Zeile 18). Sollte dies der Fall sein, bedeutet das, dass das letzte `MTemporalAnnotation`-Objekt des Elements den Beginn eines Intervalls markiert und dass dieses Intervall vor einer Annotation mit den übergebenen Werten abgeschlossen werden muss.

```
1  private void checkPreviousAnnotations(MTemporalAnnotatedEntity element, int version) {
2
3      MTemporalAnnotation lastAnnotation =
4          element.getTemporalAnnotation().get(element.getTemporalAnnotation().size()-1);
5
6      if (lastAnnotation.getVersion() != version-1) {
7        for (int i = lastAnnotation.getVersion()+1; i < version; i++) {
8          addNewAnnotation(element, lastAnnotation.getFeatureCondition(), i);
9        }
10     }
11   }
```

Codefragment 5.3.: Implementierung für das Vervollständigen eines Intervalls

Dies geschieht in Zeile 19 durch die Methode `checkPreviousAnnotations(MTemporalAnnotatedEntity element, int version)`, die auch in Codefragment 5.3 abgebildet ist. Dort wird verglichen, ob die Version des letzten `MTemporalAnnotation`-Objekts dem Wert der aktuellen Version minus eins entspricht. Falls nicht, wird für jede Version zwischen der Version des letzten `MTemporalAnnotation`-Objekts und der aktuellen Version ein `MTemporalAnnotation`-Objekt erstellt, wobei die Feature-Bedingung des letzten Objekts übernommen wird (Zeile 8).

Die Methode `addNewAnnotation(MTemporalAnnotatedEntity element, String featureCondition, int version)` erstellt ein Objekt der Klasse `MTemporalAnnotation`, setzt dessen Werte entsprechend der übergebenen Feature-Bedingung und Version und ordnet das Objekt dann dem als `MTemporalAnnotatedEntity` ebenfalls übergebenen Element zu. Zusätzlich setzt die Methode den Boolean-Wert des Attributs `intervalOpen` im `AnnotationProperties`-Objekt, welches dem Element zugeordnet ist, auf `true`.

Nachdem so, wenn nötig, sämtliche Tupel für das alte Intervall erstellt wurden, wird die Methode `addNewAnnotation(MTemporalAnnotatedEntity element, String featureCondition, int version)` noch einmal in Codefragment 5.2 in Zeile 21 aufgerufen, um ein `MTemporalAnnotation`-Objekt mit den neuen Werten zu erstellen. Dieses Objekt markiert dann wiederum den Beginn des neuen Intervalls.

Im Anschluss an das Erstellen und Annotieren der 175%-State Machine werden alle Objekte der Klassen `MAttribute`, `MEvent`, `MAction` und `MGuard`, die im Kernmodell enthalten sind, kopiert. Das bedeutet, ein neues Objekt wird erstellt und die Werte des alten Objekts werden eins zu eins

übernommen. Des Weiteren wird das neue Objekt in die jeweilige Klassen-Liste des Managers eingefügt. Wichtig ist an dieser Stelle die Reihenfolge der Erstellung der Objekte. Die Objekte der Klassen `MAttribute` und `MEvent` müssen in jedem Fall vor den Objekten der Klassen `MAction` und `MGuard` erstellt werden. Dies ist der Fall, da ein `MAction`-Objekt sowohl auf ein `MEvent`-Objekt referenzieren kann als auch in einem `MCodeFragment`-Objekt auf ein `MAttribute`-Objekt. Ein `MGuard`-Objekt kann ebenfalls in einem `MCodeFragment`-Objekt auf ein `MAttribute`-Objekt referenzieren. Eine Referenz wird im 175%-Modell erstellt, indem die ID des referenzierten Objekts im Higher-Order Delta-Modell verwendet wird, um das entsprechende 175%-Objekt aus der jeweiligen Klassen-Liste herauszusuchen und die Referenz des neuen Objekts auf dieses zu setzen. Wird in der Liste kein Objekt mit gleicher ID gefunden, wird keine Referenz erstellt. Aus diesem Grund muss das referenzierte Objekt bereits zuvor erstellt worden sein, um an dieser Stelle keinen Fehler im transformierten Modell zu erhalten.

Im nächsten Schritt wird die Root-Region des Kernmodells bearbeitet. Dazu wird sie mit Hilfe einer Factory-Klasse als 175%-Objekt erstellt, als Root-Region des 175%-Modells gesetzt und erhält die gleichen Eigenschaften wie ihr Gegenobjekt im Higher-Order Delta-Modell. Die Root-Region wird ebenfalls mit `true` und `0` annotiert. Zusätzlich wird ein `AnnotationProperties`-Objekt für die RootRegion erstellt, welches speichert, dass das Intervall der Annotation nun geöffnet ist und unter welcher Feature-Bedingung das Element zum Modell hinzugefügt wurde. Die Bedingung lautet in diesem Fall `true`, da das Element nicht durch ein Delta hinzugefügt wurde, sondern im Kernmodell enthalten ist. Das `AnnotationProperties`-Objekt wird dann in die Liste der `MTemporalAnnotatedRegion`-Klasse eingefügt.

Nachfolgend werden alle `MStateGroup`-, `MState`- und `MTransition`-Objekte, die in der Root-Region enthalten sind, nach dem gleichen Schema ebenfalls als 175%-Objekte erstellt. Wenn ein `MTemporalAnnotatedState`-Objekt erstellt wird, werden wiederum die enthaltenen `MRegion`-Objekte als 175%-Objekte überführt, und so wird rekursiv das ganze Kernmodell kopiert. Auch hierbei muss wieder darauf geachtet werden, dass Referenzobjekte erstellt wurden, bevor ihre Referenz kopiert wird. Da `MTemporalAnnotatedStateGroup`-Objekte zuerst erstellt werden, werden in diesen keine Referenzen auf `MTemporalAnnotatedState`-Objekte gesetzt. Dies findet beim Erstellen der `MTemporalAnnotatedState`-Objekte statt. Dabei werden sowohl die Referenzen der `MTemporalAnnotatedState`-Objekte auf die `MTemporalAnnotatedStateGroup`-Objekte gesetzt als auch umgekehrt. Beim Erstellen der Zustände findet wiederum kein Referenzieren auf Transitionen statt. Dies geschieht bei Erstellung der `MTemporalAnnotatedTransition`-Objekte, die erst nach den Zuständen erstellt werden. Dort werden dann erneut sowohl die Referenzen der Transitionen auf die Zustände als auch die Referenzen der Zustände auf die Transitionen gesetzt.

Nachdem so das komplette Kernmodell als 175%-Modell kopiert wurde, werden die Inhalte der Higher-Order Deltas mit der Methode `sortHODs()`, welche wieder von der Methode `transformModel(String name, String id)` aufgerufen wird, sortiert. Dabei wird für jedes Higher-Order Delta ein `SortedHOD`-Objekt erstellt, dem in der Reihenfolge der Higher-Order Deltas ein aufsteigender Integer-Wert als Version zugeordnet wird. Bei der Erstellung der `SortedHOD`-Objekte werden durch die Methode `sortCombinations(HigherOrderDelta hod)` für jedes Higher-Order Delta die enthaltenen Deltas als Kombinationen aus betroffenem Element, Delta-Operation, Element-Operation, Delta-Anwendungsbedingung und Eltern-Objekt des betroffenen Elements sortiert und gespeichert. Handelt es sich bei dem betroffenen Element um ein `MRegion`- oder `MState`-Objekt mit

untergeordneten Objekten, wird nicht für jedes dieser Objekte eine eigene Kombination angelegt. Stattdessen werden die untergeordneten Objekte bei der Auswertung der Kombination rekursiv abgearbeitet, wie es beim Kernmodell der Fall war.

Beim Erstellen der Kombinationen werden außerdem alle von einem `HigherOrderAdd` betroffenen `MStateMachineDelta`-Objekte samt enthaltenen `MStateMachineAddition`-, `MStateMachineRemoval`- und `MStateMachineModification`-Objekten kopiert und in der Delta-Klassen-Liste des Managers gespeichert. Während der Auswertung der Kombinationen werden diese Deltas angepasst, sollten an ihnen Änderungen durch `HigherOrderModify`-Objekte vorgenommen werden. Dazu werden alle `HigherOrderModify`-Objekte zusammen mit der Version des `HigherOrderDelta`-Objekts, in dem sie enthalten, als `UpdateInformation`-Objekt in der `UpdateList` gespeichert. Vor der Auswertung der Kombinationen einer Version wird überprüft, ob die `UpdateList` für diese Version `UpdateInformation`-Objekte besitzt. Ist dies der Fall, werden die `MStateMachineDelta`-Objekte in der Delta-Liste den Änderungen im `HigherOrderModify`-Objekt entsprechend angepasst. Auf diese Weise sind die Deltas immer in der aktuellsten Form gespeichert und es ist sichergestellt, dass bei der Modifikation einer Delta-Anwendungsbedingung auch wirklich nur die Annotationen der aktuell enthaltenen Elemente entsprechend angepasst werden.

Im Zuge der Speicherung der Deltas werden direkt alle Objekte, die von einer `MStateMachineAddition` betroffen sind, als 175%-Objekte erstellt und in das Modell und die Klassen-Listen eingefügt. Auf diese Weise wird sicher gestellt, dass alle Objekte bereits im 175%-Modell existieren und referenziert werden können. Selbiges gilt für Objekte, die in einer `MStateMachineAddition` enthalten sind, welche durch ein `HigherOrderModify` zu einem `MStateMachineDelta` hinzugefügt wird. Die Objekte müssen folglich beim Auswerten der Kombinationen nur noch annotiert werden. Daher werden für Objekte der Klassen `MAttribute`, `MEvent`, `MAction` und `MGuard` keine Kombinationen erstellt, da es sich bei ihnen nicht um `MTemporalAnnotatedEntity`-Objekte handelt und sie somit nicht annotiert werden müssen.

Für Element-Modifikationen kann kein Objekt der normalen `Combination`-Klasse verwendet werden, da für diese zusätzlich die Art der Modifikation gespeichert werden muss. Daher wird für Modifikationen ein Objekt der von der `Combination`-Klasse abgeleiteten `ModificationCombination`-Klasse verwendet. In dieses kann zusätzlich das `MStateMachineModification`-Objekt direkt gespeichert werden, sodass später auf die genauen Änderungen durch die Modifikation zugegriffen werden kann. Auf Basis dessen kann dann die modifizierte Version des Objektes erstellt werden, sofern sie noch nicht existiert.

Eine weitere Ausnahme stellen die Modifikationen von Anwendungsbedingungen dar. Für diese werden zwar normale `Combination`-Objekte verwendet, allerdings wird statt eines einzelnen Elements das komplette Delta, auf das sich die Änderung bezieht, in der Kombination gespeichert. Auf diese Weise ist es später möglich, auf alle im Delta enthaltenen Objekte zugreifen zu können, als auch die Bedingung bei dem in der Klassen-Liste gespeicherten Delta selbst zu ändern, sodass das Delta für den weiteren Verlauf wieder auf dem aktuellen Stand ist.

Wenn alle Higher-Order Delta-Inhalte in Kombinationen einsortiert wurden, werden die Kombinationen durch die Methode `transformDeltas()`, die von der Methode `transformModel(String name, String id)` aufgerufen wird, ausgewertet. Die Kombinationen werden beginnend mit dem `SortedHOD`-Objekt mit der niedrigsten Version in aufsteigender Reihenfolge abgearbeitet. Zunächst werden die `MStateMachineDelta`-Objekte in der Delta-Liste aktualisiert. Danach werden nach den

Regeln aus Kapitel 4.1 die neuen Feature-Bedingungen für jedes in einer Kombination enthaltene Element berechnet. Mit Hilfe der Methode `annotateElement(MTemporalAnnotatedEntity element, String featureCondition, int version)` aus Codefragment 5.2 werden die Elemente dann mit dieser Feature-Bedingung und der Version des aktuellen `SortedHOD`-Objekts annotiert.

Nachdem alle `SortedHOD`-Objekte abgearbeitet wurden, werden sämtliche Elemente in den Klassen-Listen einer annotierten Klasse sowie das `MTemporalAnnotatedStateMachine`-Objekt überprüft. Wird dabei ein Element gefunden, dessen Intervall im zugehörigen `AnnotationProperties`-Objekt geöffnet ist, das für die letzte Version aber kein `MTemporalAnnotation`-Objekt besitzt, also in dieser Version folglich nicht mehr verändert wurde, dann wird für dieses Element die Methode `checkPreviousAnnotations(MTemporalAnnotatedEntity element, int version)` aus Codefragment 5.3, welche offene Intervalle abschließt, aufgerufen. Nach Abschluss dieses Schrittes ist die Transformation des Higher-Order Delta-Modells in ein 175%-Modell abgeschlossen.

```
1  public MTemporalAnnotatedStateMachine initiateTransformation(String inputUrl,
2      String modelName, String outputFolder) {
3
4    HigherOrderDeltaRepository hodModel =
5        (HigherOrderDeltaRepository) ModelLoader.load("xmi", inputUrl);
6
7    TransformationManager manager =
8        TransformationFactory.eINSTANCE.createTransformationManager();
9    manager.setHodModel(hodModel);
10
11   String modelID = modelName + "_ID";
12   MTemporalAnnotatedStateMachine model175 =
13       manager.transformModel(modelName, modelID);
14
15   String outputUrl = outputFolder + "/175Model_" + modelName+  ".xmi";
16   ModelLoader.save(model175, null, "xmi", outputUrl);
17   System.out.println("Model saved as " +outputUrl);
18
19   return model175;
20 }
```

Codefragment 5.4.: Implementierung der Initialisierung der Transformation

Zur Durchführung der Transformation existiert im Package `de.imotep.evolution.transformation` die Klasse `RunTransformation`. In dieser Klasse ist die Methode `initiateTransformation(String inputUrl, String modelName, String outputFolder)` enthalten, welche in Codefragment 5.4 abgebildet ist. Diese Methode lädt zunächst mit Hilfe der Klasse `ModelLoader`, die ebenfalls Teil des selben Packages ist, in Zeile 5 eine Datei mit einem Higher-Order Delta-Modell anhand der Speicheradresse der Datei ein. Danach wird ein `TransformationManager`-Objekt erstellt und diesem das geladene `HigherOrderDeltaRepository`-Objekt übergeben. Daraufhin wird aus dem Modell-Namen, der bei Aufruf der Methode `initiateTransformation` übergeben wurde, die Modell-ID gebildet. Diese könnte wahlweise allerdings auch manuell vergeben werden. Anschließend wird in Zeile 16 die zuvor beschriebene Methode `transformModel(String name, String id)`, welche die Transformation steuert, aufgerufen. Diese Methode gibt anschließend das fertige 175%-Modell zurück. Dieses Modell wird abschließend mit Hilfe der Klasse `ModelLoader` in dem Ordner

gespeichert, dessen Speicheradresse in dem String `outputFolder` übergeben wurde. Abschließend wird das Modell als Rückgabewert der Methode zurückgegeben. Mit Hilfe dieser Methode kann die Transformation in andere Programme eingebettet oder auch einfach nur unabhängig ausgeführt werden.

Genauso wie bei der Versionsableitung existieren in der Klasse `RunTransformation` weitere ähnliche Methoden zur Ausführung der Transformation. Diese sind die Methoden `initiateTransformation(String inputUrl, String modelName)`, falls das Ausgabe-Modell nicht gespeichert werden soll, die Methode `initiateTransformation(HigherOrderDeltaRepository hodModel, String modelName, String outputFolder)`, falls das Eingabe-Modell nicht geladen werden soll, und `initiateTransformation(HigherOrderDeltaRepository hodModel, String modelName)`, falls weder das Eingabe-Modell geladen noch das Ausgabe-Modell gespeichert werden soll.

Für die Transformation wurde ebenfalls ein Wizard erstellt, der über *New > Other...* im Untermenü *Model Transformation* aufgerufen werden kann. Der Wizard zeigt den gleichen Aufbau wie der Wizard für die Versionsableitung, jedoch muss hier statt einer Version ein Name für das zu erzeugende 175%-Modell eingegeben werden. Der Wizard ruft die Methode `transform(String inputUrl, String modelName, String outputFolder)` der Klasse `RunTransformation` mit den eingegebenen Werten auf. Diese Methode verhält sich wie die Methode in Codefragment 5.4, besitzt allerdings keinen Rückgabewert.

Im folgenden Kapitel wird die Erweiterung der bestehenden Slicing-Funktionalität auf 175%-Slicing beschrieben.

5.5. Erweiterung auf 175%-Slicing

Wie in Kapitel 5.2 bereits erwähnt, bestand bereits eine Funktionalität zum Slicing von normalen State Machines und 150%-State Machines durch die Plug-ins `de.imotep.slicing` und `de.imotep.slicing.dependency`. Daher werden zunächst einige Teile der bestehenden Implementierung aufgezeigt und nachfolgend die auf Basis dessen vorgenommenen Erweiterungen.

Bestehende Implementierung

Der grobe Arbeitsablauf bei der Verwendung der Slicing-Funktionalität ist in Abbildung 5.19 dargestellt. Dieser ändert sich auch bei der Erweiterung der Implementierung nicht und kann genauso weiter durchgeführt werden.

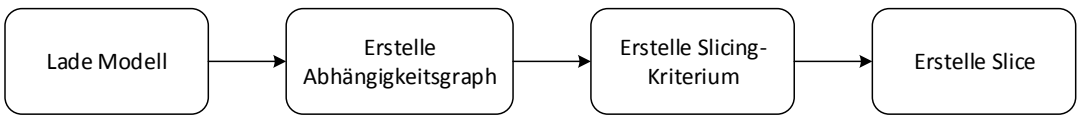


Abbildung 5.19.

Für das Slicing wird zunächst durch eine Factory-Klasse ein Slicing Manager erstellt. Weiterhin wird ein Modell einer State Machine oder einer 150%-State Machine aus einer XMI-Datei geladen, auf deren Basis dann ein Abhängigkeitsgraph für die enthaltenen Elemente (s. Kapitel 2.3) durch den Manager erzeugt wird. Zusätzlich wird eine Slicing-Strategie ausgewählt (vorwärts oder rückwärts) und das gewünschte Kriterium für den Slice erstellt. Ein normales Kriterium besitzt lediglich

ein Slicing-Ziel, welches das Element bezeichnet, für das der Slice erstellt werden soll. Dieses Element kann entweder ein Zustand oder eine Transition sein und ist das erste im Slice enthaltene Element. Ein 150%-Kriterium ist von einem normalen Kriterium abgeleitet und enthält zusätzlich einen aussagenlogischen Ausdruck über Feature-Parameter in Form eines Strings zur Definition der gewünschten Feature-Konfiguration.

Nachdem alle Eigenschaften für den zu erstellenden Slice definiert wurden, werden ausgehend vom Slicing-Ziel weitere Elemente rekursiv zum Slice hinzugefügt. Hierbei werden 150%-Kriterium und annotierte Elemente zunächst wie ein normales Kriterium und normale Elemente behandelt, um generisch arbeiten zu können. Sobald allerdings für ein annotiertes Element festgestellt wurde, dass es aufgrund der Abhängigkeiten zum Slice hinzugefügt werden soll, muss zusätzlich die Annotation des Elements überprüft werden. Daher gibt es an den drei Stellen im Programm, an denen ein Element zum Slice hinzugefügt werden soll, eine Überprüfung, ob es sich bei dem gewählten Kriterium um ein 150%-Kriterium handelt. Da 150%-Kriterien nur in Zusammenhang mit 150%-State Machines vom Programm geduldet werden, muss es sich folglich bei dem hinzuzufügenden Element um ein annotiertes Element handeln, wenn das Kriterium ein 150%-Kriterium ist.

```
1  if (criterium instanceof MSliceCriterium150) {
2    MSliceCriterium150 criterium150 = (MSliceCriterium150) criterium;
3
4    if (dependentNode instanceof MStateNode) {
5      MAnnotatedState annState =
6          (MAnnotatedState)((MStateNode)dependentNode).getState();
7      if (!isCompatible(annState.getAnnotation(), criterium150)) {
8        continue;
9      }
10   }
11
12   else {
13     MAnnotatedTransition annTransition =
14         (MAnnotatedTransition) ((MTransitionNode)dependentNode).getTransition();
15     if (!isCompatible(annTransition.getAnnotation(), criterium150)) {
16       continue;
17     }
18   }
19 }
```

Codefragment 5.5.: Implementierung zur Überprüfung von 150%-Elementen

Codefragment 5.5 zeigt stellvertretend eine dieser Stellen. Über die if-Bedingung wird abgefragt, ob es sich um ein 150%-Kriterium handelt. Ist dies der Fall, werden das Kriterium und das Element unter Betrachtung in ein 150%-Kriterium und ein annotiertes Element gecastet. In dieser Form kann dann die Methode `isCompatible(String annotation, MSliceCriterium150 c)` aufgerufen werden, welche überprüft, ob die Annotation des Elements und der Ausdruck des Kriteriums miteinander vereinbar sind. Die Implementierung dieser Methode ist in Codefragment 5.6 abgebildet. Die Methode `isCompatible(String annotation, MSliceCriterium150 c)` überprüft, ob in der Annotation eine Formel gespeichert ist, und wandelt diese Formel von einem String in eine Boolean Expression um. Sollte keine Formel gespeichert sein, wird `true` zurückgegeben. Der Ausdruck

im Slicing-Kriterium wird ebenfalls in eine Boolean Expression umgewandelt. Für diese beiden Expressions wird dann die Methode `isComparable(BooleanExpression be1, BooleanExpression be2)` der Klasse `ConditionComparer` aufgerufen, welche die beiden Expressions auswertet und `true` zurückgibt, wenn ihre Aussagen miteinander vereinbar sind, oder false zurückgibt, wenn dies nicht der Fall ist (vgl. Kamischke [20]). Dieser Boolean-Wert wird direkt als return-Wert für die Methode `isCompatible(String annotation, MSliceCriterium150` zurückgegeben. Bei `true` wird das Element in den Slice aufgenommen, bei `false` nicht.

```
1  private boolean isCompatible(String annotation, MSliceCriterium150 c){
2    BooleanExpression bexp = null;
3
4    if(annotation.length()>0){
5      bexp= new BooleanExpression(annotation);
6    }
7    else{
8      return true;
9    }
10
11   try {
12     return
13       ConditionComparer.isComparable(new BooleanExpression(c.getExpression()), bexp);
14   } catch (TimeoutException e) {
15     ...
16     return false;
17   }
18 }
```

Codefragment 5.6.: Implementierung zum Vergleich von 150%-Annotation und -Kriterium

Nachdem nun einige ausgewählte Stellen der Implementierung vorgestellt wurden, werden im Folgenden die Erweiterungen dieser Stellen beschrieben.

Erweiterte Implementierung

Die Implementierung des 175%-Slicings wird direkt im bestehenden Plug-in `de.imotep.slicing` umgesetzt. Dazu müssen beide in den Codefragmenten gezeigten Stellen sowie zwei weitere Stellen, die analog zu Codefragment 5.5 aufgebaut sind, für 175%-Modelle erweitert und angepasst werden.

```
1 if ((criterium instanceof MSliceCriterium150) &&
2     !(criterium instanceof MSliceCriterium175)) {
3   ...
4 }
```

Codefragment 5.7.: Implementierung zur Typüberprüfung des Slicing-Kriteriums

Codefragment 5.7 zeigt die Veränderung, die an Codefragment 5.5 vorgenommen wurde. Die ursprüngliche Bedingung musste an dieser Stelle erweitert werden, da bei der einfachen Abfrage, ob es sich um ein 150%-Kriterium handelt, nicht mehr eindeutig geschlussfolgert werden kann, ob es sich tatsächlich nur um ein 150%-Kriterium handelt oder ob es ein 175%-Kriterium ist. Dies liegt

daran, dass die Klasse `MSliceCriterium175` von der Klasse `MSliceCriterium150` abgeleitet wurde, und jedes 175%-Kriterium somit auch ein Objekt der `MSliceCriterium150`-Klasse ist. Aus diesem Grund wird an dieser Stelle sowie an zwei weiteren Stellen, welche analog aufgebaut sind, nun immer abgefragt, ob es sich um ein 150%-Kriterium, aber gleichzeitig nicht um ein 175%-Kriterium handelt.

Da der auszuführende Inhalt der if-Bedingung speziell für 150%-Elemente gilt, musste ein analoges Codefragment für 175%-Elemente erstellt werden, um für diese ebenfalls überprüfen zu können, ob die Annotation dem Kriterium entspricht. Nur dann können die Elemente wieder zum Slice hinzugefügt werden. Codefragment 5.8 enthält diese neue Implementierung für die Überprüfung von 175%-Elementen. Die else-if-Bedingung schließt direkt an die if-Bedingung aus Codefragment 5.7 an. Handelt es sich bei dem eingegebenen Kriterium um ein 175%-Kriterium, dann werden das bis dorthin als einfaches Kriterium behandelte Kriterium und die als einfache Elemente behandelten Elemente sowohl in ein 175%-Kriterium als auch in 175%-Elemente gecastet. In dieser Form kann die Überprüfung der Annotation des Elements auf Übereinstimmung mit dem Slicing-Kriterium geprüft werden.

```
1  else if (criterium instanceof MSliceCriterium175) {
2    MSliceCriterium175 criterium175 = (MSliceCriterium175) criterium;
3
4    if (dependentNode instanceof MStateNode) {
5      MTemporalAnnotatedState tempAnnState =
6          (MTemporalAnnotatedState)((MStateNode)dependentNode).getState();
7      if (!isCompatible175(tempAnnState.getTemporalAnnotation(), criterium175)) {
8        continue;
9      }
10   }
11
12   else {
13     MTemporalAnnotatedTransition tempAnnTransition =
14       (MTemporalAnnotatedTransition)((MTransitionNode)dependentNode).getTransition();
15     if (!isCompatible175(tempAnnTransition.getTemporalAnnotation(), criterium175)) {
16       continue;
17     }
18   }
19 }
```

Codefragment 5.8.: Implementierung zur Überprüfung von 175%-Elementen

Diese Überprüfung findet durch die neue Methode `isCompatible175(EList<MTemporalAnnotation> annotations, MSliceCriterium175 criterium)` statt, welche für das 175%-Slicing neu implementiert wurde. Diese Methode ist in Codefragment 5.9 dargestellt. Der Methode wird eine Liste mit den `MTemporalAnnotation`-Objekten eines Elements sowie das 175%-Kriterium übergeben. Die übergebene Liste von Annotations-Objekten wird zunächst in eine `CopyOnWriteArrayList` gespeichert. Diese ist im Gegensatz zu einer normalen Liste während der Iteration veränderbar, sodass während des Iterierens Objekte daraus entfernt werden können, ohne dass eine Exception ausgelöst wird. Über diese Liste wird dann iteriert und für jedes Objekt verglichen, ob die enthaltene Version einer der Versionen des 175%-Kriteriums gleicht (Zeile 11). Stimmt keine der Versionen mit

der Version des Objekts überein, wird das Objekt aus der Liste entfernt (Zeile 17). So verbleiben am Ende nur die Objekte in der Liste, deren Versionen dem Kriterium entsprechen.

```
1  private boolean isCompatible175(EList<MTemporalAnnotation> annotations,
2      MSliceCriterium175 criterium) {
3
4    CopyOnWriteArrayList<MTemporalAnnotation> remainingAnnotations =
5        new CopyOnWriteArrayList<MTemporalAnnotation>();
6    remainingAnnotations.addAll(annotations);
7
8    for (MTemporalAnnotation annotation : remainingAnnotations) {
9      boolean foundEqualVersion = false;
10     for (Integer critVersion : criterium.getVersions()) {
11       if (critVersion == annotation.getVersion()) {
12         foundEqualVersion = true;
13         break;
14       }
15     }
16     if (!foundEqualVersion) {
17       remainingAnnotations.remove(annotation);
18     }
19   }
20
21   for (MTemporalAnnotation annotation : remainingAnnotations) {
22     boolean featureConditionCompatible =
23         isCompatible(annotation.getFeatureCondition(), criterium);
24     if (!featureConditionCompatible) {
25       remainingAnnotations.remove(annotation);
26     }
27   }
28
29   if (remainingAnnotations.isEmpty()) {
30     return false;
31   }
32   else {
33     return true;
34   }
35 }
```

Codefragment 5.9.: Implementierung zum Vergleich von 175%-Annotation und -Kriterium

Im nächsten Schritt wird erneut über die Liste mit den restlichen Annotations-Objekten iteriert. An dieser Stelle wird die im Objekt enthaltene Feature-Bedingung auf Kompatibilität mit der Feature-Bedingung des Kriteriums geprüft, indem die Methode `isCompatible(String annotation, MSliceCriterium150 c)` für 150%-Modelle, die in Codefragment 5.6 beschrieben ist, aufgerufen wird (Zeile 23). Liefert die Methode `false` zurück, weil die Feature-Bedingungen einander nicht entsprechen, wird das `MTemporalAnnotation`-Objekt aus der Liste entfernt. Ist die Liste nach Abschluss der Iteration leer, existiert kein Tupel in der Annotation des Elements, dessen Annotation überprüft werden sollte, das die Bedingungen des Kriteriums erfüllt. In diesem Fall liefert die Methode `isCompatible175(EList<MTemporalAnnotation> annotations, MSliceCriterium175`

criterium) false zurück und das Element wird nicht zum Slice hinzugefügt. Enthält die Liste noch mindestens ein Objekt, ist die Annotation mit dem Kriterium kompatibel, und die Methode liefert true zurück, sodass das Element zum Slice hinzugefügt wird.

```
1  public static void main(String[] args) {
2    MSlicingManager manager = SlicingFactory.eINSTANCE.createMSlicingManager();
3    MSlicingStrategy strategy =
4        SlicingFactory.eINSTANCE.createMSlicingStrategyForward();
5    manager.setSlicingStrategy(strategy);
6
7    MTemporalAnnotatedStateMachine temporalStateMachine =
8        (MTemporalAnnotatedStateMachine) Loader.load("xmi","input/Testmodel_Wiper.xmi");
9    MDependencyGraph depGraph = manager.buildDependencyGraph(temporalStateMachine);
10
11   MState initial = ((MStateNode) depGraph.findNode("Wiper")).getState();
12   String featureConfiguration =
13       "HighQualityWiper&&HighQualitySensor&&!Clean&&!Intensity";
14   BasicEList<Integer> versions = new BasicEList<Integer>();
15   versions.add(2);
16   versions.add(3);
17   versions.add(4);
18   MSliceCriterium175 sliceCriterium =
19       manager.createSliceCriterium175(initial, featureConfiguration, versions);
20
21   MSlice slice = manager.createSlice(sliceCriterium, depGraph);
22 }
```

Codefragment 5.10.: Durchführen des Slicings

Mit Hilfe dieser Änderungen konnte das bestehende Slicing-Tool folglich so erweitert werden, dass es zusätzlich zum normalen und zum 150%-Slicing auch für 175%-Slicing funktioniert. Codefragment 5.10 zeigt die beispielhafte Durchführung eines Slicings. Dazu wird zuerst in Zeile 2 ein Objekt der Klasse MSlicingManager erstellt sowie in Zeile 4 ein Objekt der Klasse MSlicingStrategy, welches als Slicing-Strategie des Managers gesetzt wird. Danach wird eine 175%-State Machine aus einer XMI-Datei eingelesen (Zeile 8), für die daraufhin ein Abhängigkeitsgraph erstellt wird. Anschließend wird aus dem Abhängigkeitsgraph anhand des Namens das Element herausgesucht, welches als Element für das Slicing-Kriterium dienen soll (Zeile 11). Weiterhin werden ein String mit der gewünschten Feature-Konfiguration als auch eine Liste von Integer-Werten für die gewünschten Versionen erstellt. Aus Element, Feature-Konfiguration und Versionsliste wird dann in Zeile 19 das 175%-Slicing-Kriterium angelegt. Zuletzt wird auf Basis des Kriteriums und des Abhängigkeitsgraphen der Slice für das 175%-Modell erzeugt.

Im folgenden Kapitel sollen die implementierten Funktionalitäten hinsichtlich ihres Zeitverhaltens evaluiert werden.

6 Evaluation

In diesem Kapitel erfolgt die Evaluation des Zeitverhaltens der implementierten Funktionalitäten. Das Ziel ist hierbei zum einen, eine grobe Aussage über die Laufzeit der unterschiedlichen Funktionalitäten zu erhalten. Des Weiteren besteht ein kompletter Programmdurchlauf einer Funktionalität aufgrund der Verwendung von Modellen aus verschiedenen Phasen, wie beispielsweise dem Laden oder Speichern der Modelle zusätzlich zur reinen Berechnung der Transformation oder der Versionsableitung. Hierbei ist die Identifikation der Phasen mit der höchsten Laufzeit von besonderem Interesse, um möglicherweise Stellen zu bestimmen, an denen Optimierungspotenzial für eine geringere Laufzeit besteht. Ferner existieren zwei unterschiedliche Möglichkeiten einen Slice aus einem 175%-Modell für ein Kriterium mit nur einer Version anzufertigen. Einerseits kann direkt das 175%-Slicing verwendet werden, andererseits besteht die Möglichkeit zuerst eine Version aus dem 175%-Modell abzuleiten und auf diesem 150%-Modell dann ein 150%-Slicing durchzuführen. An dieser Stelle soll die effizientere der beiden Möglichkeiten bestimmt werden, um eine Empfehlung geben zu können, welches Vorgehen in diesem Fall verwendet werden sollte. Folglich stellen sich für die Evaluation folgende Forschungsfragen:

- Was sind die durchschnittlichen Laufzeiten der verschiedenen Funktionalitäten?
- Welche Phasen der Programmausführung benötigen den größten Teil der Laufzeit?
- Hat das 175%-Slicing oder eine Kombination aus Versionsableitung und 150%-Slicing eine höhere Laufzeit?

Die Evaluation des Zeitverhaltens der implementierten Funktionalitäten erfolgt mit Hilfe der Modelle von vier delta-orientierten Fallstudien mit Evolutionshistorie [29]. Die vier Fallstudien beschreiben einen Verkaufsautomaten, eine Scheibenwischanlage, eine Minenpumpanlage [11] und ein Body Comfort System [26]. Für jede der Fallstudien liegt zu Beginn ein Higher-Order Delta-Modell vor [29]. Diese Higher-Order Delta-Modelle werden durch die Funktionalität des Plug-ins `de.imotep.evolution.transformation` (s. Kapitel 5.4) in 175%-Modelle umgewandelt. Auf diesen 175%-Modellen kann dann sowohl eine Versionsableitung von 150%-Modellen durch das Plugin `de.imotep.evolution.temporalAnnotation` als auch 175%-Slicing durch das Plug-in `de.imotep.slicing` stattfinden.

Jede dieser drei Funktionalitäten wird auf ihr Zeitverhalten hin untersucht, indem sie mehrmals auf die Modelle der verschiedenen Fallstudien angewendet wird. Dabei muss sich eine repräsentative Anzahl von Durchläufen ergeben, die ein aussagekräftiges Durchschnittsergebnis erlaubt. Zusätzlich wird auf den abgeleiteten 150%-Modell-Versionen das bereits zuvor bestehende 150%-Slicing angewendet. Auf diese Weise können die Ergebnisse für die Versionsableitung und das 150%-Slicing miteinander kombiniert werden. Dies ermöglicht einen Vergleich des Zeitverhaltens von der Kombination aus Versionsableitung und 150%-Slicing mit dem Zeitverhalten des 175%-Slicings.

Die Werte für die Laufzeit werden ermittelt, indem Test-Methoden für das Ausführen der Funktionalitäten verwendet werden. In diesen Test-Methoden werden an gewissen Punkten Zeitstempel mit der aktuellen Zeit in Millisekunden gesetzt, die als String in einer Datei gespeichert werden. Codefragment 6.1 zeigt die Test-Methode für die Transformation.

```
1  public String testTransformation(String inputUrl, String modelName) {
2    String data = "";
3
4    //tag: load
5    data = data + Calendar.getInstance().getTimeInMillis() + "#";
6    HigherOrderDeltaRepository hodModel =
7        (HigherOrderDeltaRepository) ModelLoader.load("xmi", inputUrl);
8
9    //tag: transform
10   data = data + Calendar.getInstance().getTimeInMillis() + "#";
11   TransformationManager manager =
12       TransformationFactory.eINSTANCE.createTransformationManager();
13   manager.setHodModel(hodModel);
14   String modelID = modelName + "_ID";
15   MTemporalAnnotatedStateMachine model175 =
16       manager.transformModel(modelName, modelID);
17
18   //tag: save
19   data = data + Calendar.getInstance().getTimeInMillis() + "#";
20   String outputUrl = "output/175Model_" + modelName+  ".xmi";
21   ModelLoader.save(model175, null, "xmi", outputUrl);
22
23   //tag: end
24   data = data + Calendar.getInstance().getTimeInMillis() + "#;";
25   return data;
26 }
```

Codefragment 6.1.: Zeitstempel für die Evaluation der Transformation

Hierbei ergeben sich vier Punkte, an denen Zeitstempel gesetzt wurden (Zeilen 5, 10, 19 und 24). Der erste Zeitstempel bezeichnet den Start der Funktionalität, bevor das Modell geladen wird. Der zweite Zeitstempel kennzeichnet den Zeitpunkt, ab dem alle notwendigen Werte für den Aufruf der Transformation gesetzt werden, wie das Erstellen eines `TransformationManager`-Objekts und das Setzen des Higher-Order Delta-Modells sowie das Erstellen einer ID aus dem übergebenen Modell-Namen. Nach Abschluss der Transformation erfolgt der dritte Zeitstempel. Nach diesem wird das Modell gespeichert, wobei direkt nach der Speicherung der vierte Zeitwert gespeichert wird. Die Zeitwerte in Millisekunden werden, abgegrenzt durch das Trennzeichen „#“, in einen String gespeichert, der am Ende der Methode zurückgegeben wird. Die aufrufende Methode speichert den String in einer Text-Datei. Sämtliche Strings für die Ausführung der selben Funktionalität mit gleichen Eingabewerten - also etwa die Transformation mit gleichem Eingabe-Modell - werden nacheinander in der selben Datei gespeichert. Die Strings für die verschiedenen Durchläufe sind dabei durch das Trennzeichen „; “abgegrenzt (Zeile 24).

Nachdem so die Daten für mehrere Testdurchläufe erzeugt wurden, werden die Daten ausgewertet, indem der String aus der Datei eingelesen und die Zeiten für die unterschiedlichen Phasen be-

rechnet werden. Für die Gesamtlaufzeit wird der Wert des ersten Zeitstempels vom Wert des letzten Zeitstempels abgezogen. Für die Ladezeit wird der erste vom zweiten Zeitstempel und für die Transformationszeit der zweite Zeitstempel vom dritten abgezogen. Ebenso wird für die Speicherzeit der Wert des dritten Zeitstempels vom Wert des vierten Zeitstempels abgezogen. Diese Zeitwerte für die drei Phasen und die Gesamtlaufzeit werden in ein zweidimensionales Array gespeichert. Eine Dimension enthält alle Testdurchläufe, während die andere Dimension die verschiedenen Phasen eines Testdurchlaufs speichert. Mit Hilfe dieses Arrays werden dann die Durchschnittswerte für jede Phase berechnet. Diese Durchschnittswerte können dann für die vier Fallstudien verglichen werden.

Die Generierung von Zeitwerten für die Versionsableitung, das 150%-Slicing und das 175%-Slicing erfolgt nach dem gleichen Prinzip. Auch hier werden für alle drei Funktionalitäten vier Zeitstempel gesetzt. Für die Versionsableitung sind diese ebenfalls am Start der Funktionalität, dann nach dem Laden des Modells vor der eigentlichen Ableitungsphase, sowie nach der Ableitungsphase vor dem Speichern, und am Ende nach dem Speichern. Sowohl für das 150%- als auch das 175%-Slicing entfällt die Speicherphase, da der Slice nicht gespeichert wird. Da der Verlauf beider Slicing-Arten gleich ist, erhalten beide die gleichen Zeitstempel um einen Vergleich zu erlauben. Diese liegen zu Beginn vor dem Laden des Modell, dann nach dem Laden vor dem Erstellen des Abhängigkeitsgraphen, nach Erstellen des Graphen vor dem eigentlichen Erstellen des Slices, und nachdem der Slice fertiggestellt wurde.

Im Folgenden werden die verwendeten Fallstudien kurz vorgestellt, um eine Basis für Vergleiche zwischen den Zeiten, die sich bei der Verwendung der Funktionalitäten für die verschiedenen Fallstudien ergeben, zu schaffen. Danach folgt die Vorstellung und Auswertung der Durchschnittslaufzeiten der verschiedenen Testläufe.

6.1. Fallstudien

Die vier Fallstudien werden hier kurz beschrieben und hinsichtlich Modellgröße und -komplexität in einen Zusammenhang gebracht, damit Vergleichswerte möglich sind.

Verkaufsautomat Die Fallstudie des Verkaufsautomaten (engl. *vending machine* (VM)) beschreibt einen Verkaufsautomaten, der drei verschiedene Getränke, zwei Währungen und drei verschiedene Milchsorten anbietet, sowie einen Milchzähler, eine Milchanzeige und einen Ton bei Getränkeausgabe. Die Evolutionshistorie dieser Fallstudie erstreckt sich über acht Versionen. Am Ende der Evolutionshistorie besitzt die Softwareproduktlinie des Verkaufsautomaten insgesamt 17 Features inklusive Wurzel-Feature, mit denen 90 verschiedene Produktvarianten gebildet werden können. Das 175%-Modell des Verkaufsautomaten, das sich nach der Transformation aus dem Higher-Order Delta-Modell ergibt, enthält insgesamt 857 Elemente. Davon sind 31 Elemente Zustände, 93 Elemente sind Transitionen und fünf Elemente sind Regionen. Der Rest verteilt sich auf Attribute, Actions, Events, Guards, Codefragmente und Annotationen, wobei die Annotationen mit 532 Elementen einen wesentlichen Teil ausmachen. Die Tiefe der State Machine liegt bei drei Regionen, wobei die Wurzelregion der State Machine das erste Regionslevel verkörpert. Die maximale Anzahl an parallelen Regionen auf einem Level beträgt drei Regionen.

Scheibenwischanlage Die Fallstudie der Scheibenwischanlage (engl. *wiper*) beschreibt eine Scheibenwischanlage, die einen Sensor und einen Wischer in zwei unterschiedlichen Qualitätsstufen,

eine permanente Wischfunktion, eine Scheibenwischfunktion und einen Scheibenwischerschutz im Frostfall bietet. Die Evolutionshistorie der Fallstudie umfasst sechs Versionen und am Ende der Historie besitzt die Produktlinie insgesamt 14 Features. Mit diesen lassen sich 84 verschiedene Produktvarianten bilden. Das 175%-Modell dieser Fallstudie besitzt insgesamt 637 Elemente, von denen 24 Zustände, 68 Transitionen und sechs Regionen sind. Die Tiefe der State Machine beträgt zwei Level von Regionen, die maximale Anzahl an paralellen Regionen liegt hier bei fünf Regionen.

Minenpumpanlage Die Fallstudie der Minenpumpanlage charakterisiert eine Minenpumpanlage (engl. *mine pump system* (MP)), die eine Wasserstandregulierung, eine Methanerkennung, eine manuelle Kommandofunktion und eine Luftüberprüfung umfasst. Die Evolutionshistorie dieser Fallstudie enthält vier Versionen. Am Ende der Versionshistorie besitzt die Softwareproduktlinie der Minenpumpanlage insgesamt neun Features, die zur Bildung von 48 verschiedenen Produktvarianten beitragen. Das 175%-Modell der Minenpumpanlage enthält 728 Elemente, davon 45 Zustände, 76 Transitionen und elf Regionen. Die State Machine hat eine Tiefe von drei Regionen. Hierbei ergeben sich sechs Regionen als maximale Anzahl von parallelen Regionen auf einem Level.

Body Comfort System Die Fallstudie des Body Comfort Systems (BCS) beschreibt unterschiedliche Komfortfunktionen eines Fahrzeugs, wie etwa die bereits beschriebene Scheibenwischanlage, elektrische Fensterheber, ein Sicherheitssystem mit Zentralverriegelung, Alarmanlage und Funkschlüssel, beheizbare Außenspiegel und Fensterscheiben, einen automatisch verstellbaren Fahrersitz, eine Mensch-Maschine-Schnittstelle mit verschiedenen LEDs und automatisches Licht. Die Evolutionshistorie des Body Comfort Systems erstreckt sich über fünf Versionen und am Ende der Historie umfasst die Produktlinie dieser Fallstudie insgesamt 48 Features. Für das Body Comfort System ergeben sich mehr als 11000 verschiedene Produktvarianten. Das 175%-Modell dieser Fallstudie hat eine Größe von 3412 Elementen. Von diesen sind 216 Elemente Zustände, 304 Elemente sind Transitionen und 75 Elemente sind Regionen, wobei die State Machine eine Tiefe von sechs Regionen umfasst. Die maximale Anzahl an parallelen Region auf einem Level beträgt an dieser Stelle zwanzig Regionen.

	Wiper	MP	VM	BCS
Modell-Größe in Anzahl Elemente	637	728	857	3412
davon Zustände	24	45	31	216
davon Transitionen	68	76	93	304
davon Regionen	6	11	5	75
Modell-Tiefe in Level von Regionen	2	3	3	6
Max. Anzahl an parallelen Regionen	5	6	3	20

Abbildung 6.1.: Vergleichsfaktoren für die 175%-Modelle der Fallstudien

Die Datei-Größe, Tiefe und maximale Anzahl an parallelen Regionen der 175%-State Machines der verschiedenen Fallstudien sind nochmals in der Tabelle in Abbildung 6.1 zusammengefasst. Dabei sind die Fallstudien nach aufsteigender Datei-Größe sortiert.

Im Folgenden werden die Ergebnisse erläutert und ausgewertet, die sich für das Zeitverhalten der verschiedenen Funktionalitäten der Implementierung ergeben.

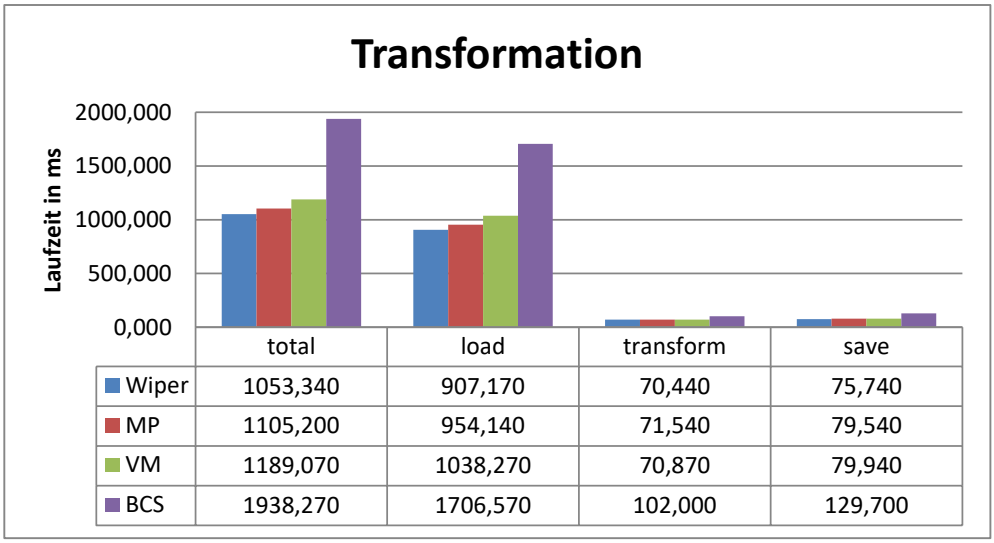
6.2. Zeitverhalten der Funktionalitäten

Für die Evaluation des Zeitverhaltens wurden die vier Funktionalitäten Transformation, Versionsableitung, 150%-Slicing und 175%-Slicing getestet. Dabei sollte jede der Funktionalitäten so oft ausgeführt werden, dass ein repräsentativer Durchschnittswert erzielt werden kann. Hierbei wurde eine Anzahl von 30 Testdurchläufen für jeden Versuch mit gleichen Eingaben ausgewählt. Diese Anzahl erlaubt eine angemessene Mittelung, selbst wenn einige Ausschläge durch beispielsweise Messfehler entstehen sollten.

Folglich wurde die Transformation von Higher-Order Delta-Modellen in 175%-Modelle für jede Fallstudie 30 mal durchgeführt. Bei der Versionsableitung wurde für jede Fallstudie aus dem 175%-Modell jede Version als 150%-Modell 30 mal abgeleitet. Auf jedes so entstandene 150%-Modell der verschiedenen Fallstudien wurde wiederum 30 mal das 150%-Slicing angewendet. Ebenso wurde auf jedes 175%-Modell das 175%-Slicing angewendet, wobei das Slicing für jede Version der Fallstudie als Kriterium jeweils 30 mal ausgeführt wurde. Für das Body Comfort System musste die Anzahl sowohl der 150%- als auch der 175%-Slicing-Durchläufe reduziert werden, da die Laufzeit hier deutlich erhöht war.

Transformation

Die durchschnittliche Laufzeit der Transformation für die verschiedenen Fallstudien ist in Abbildung 6.2 abgebildet. Der Wert `total` bezeichnet die Gesamtlaufzeit der Transformation, während `load` die Zeit zum Laden der Modelldatei, `transform` die tatsächliche Rechenzeit für die Transformation des Modells und `save` die Zeit zum Speichern des Modells in einer neuen XMI-Datei beschreibt. Alle Laufzeiten sind in Millisekunden angegeben.



	total	load	transform	save
Wiper	1053,340	907,170	70,440	75,740
MP	1105,200	954,140	71,540	79,540
VM	1189,070	1038,270	70,870	79,940
BCS	1938,270	1706,570	102,000	129,700

Abbildung 6.2.: Zeitverhalten Transformation

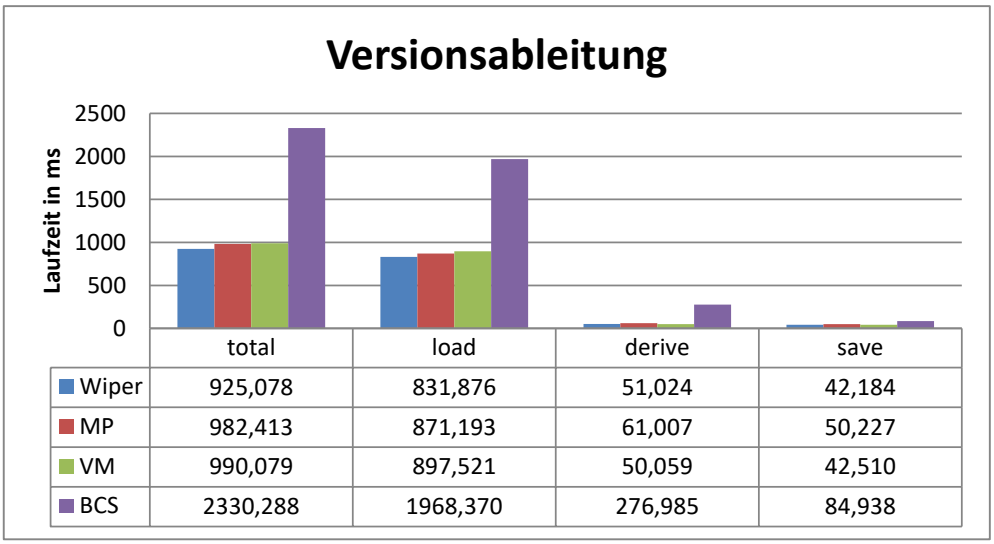
Abbildung 6.2 zeigt unter Einbezug von Abbildung 6.1, dass die Gesamtlaufzeit der Transformation mit steigender Modell-Größe der jeweiligen Fallstudie zunimmt. Für die Fallstudien Scheibenwischanlage, Minenpumpanlage und Verkaufsautomat liegt die Gesamtlaufzeit knapp über einer Sekunde, während sie für das Body Comfort System mit knapp unter zwei Sekunden fast doppelt

so lang ist. Dieses Verhältnis spiegelt ebenfalls Parallelen zur Modell-Größe wieder, da diese für das Body Comfort System vier bis fünf mal so hoch wie für die anderen drei Fallstudien ist. Die Korrelation von Laufzeit und Modell-Größe lässt sich vermutlich darauf zurückführen, dass bei größeren Modellen auch mehr Objekte geladen, kopiert und gespeichert werden müssen.

Weiterhin verdeutlicht die Abbildung, dass das Laden der Modell-Datei einen wesentlichen Teil der Gesamtzeit der Funktionalität in Anspruch nimmt. Dieser Anteil liegt bei 86,1-88,1% der Gesamtlaufzeit. Die reine Rechenzeit für die Transformation nimmt hingegen mit 5,2-6,7% den geringsten Teil der Gesamtlaufzeit in Anspruch. Auch das Speichern des Modells dauert mit 6,6-7,1% der Gesamtlaufzeit nur unwesentlich länger als die Transformation selbst.

Versionsableitung

Das Zeitverhalten für die Ableitung einer 150%-Version aus einem 175%-Modell ist in Abbildung 6.3 dargestellt. Hierbei wurden zunächst für alle vier Fallstudien je 30 Testläufe für die Ableitung jeder Version durchgeführt. Somit ergeben sich für die Scheibenwischanlage fünf, für die Minenpumpanlage drei, für den Verkaufsautomaten sieben und für das Body Comfort System vier Durchschnittswerte. Um für jede Fallstudie einen repräsentativen Mittelwert zu erhalten, anhand dessen die verschiedenen Fallstudien verglichen werden können, wurden diese Durchschnittswerte für die verschiedenen Versionen erneut gemittelt. Auf diese Weise ergibt sich für jede Fallstudie genau ein Wert für jede Phase, welcher dann mit den Werten der anderen Fallstudien verglichen werden kann. Ein Vergleich für jede einzelne Version wäre an dieser Stelle nicht sinnvoll gewesen, da einserseits jede Fallstudie eine unterschiedliche Anzahl von Versionen aufweist und andererseits die verschiedenen Evolutionsszenarien keine geeigneten Kriterien besitzen, über die allgemeingültig miteinander verglichen werden könnte. Eine Mittelung über sämtliche Versionen hingegen ermöglicht einen Vergleich der Werte für die kompletten Evolutionshistorien miteinander.



	total	load	derive	save
Wiper	925,078	831,876	51,024	42,184
MP	982,413	871,193	61,007	50,227
VM	990,079	897,521	50,059	42,510
BCS	2330,288	1968,370	276,985	84,938

Abbildung 6.3.: Zeitverhalten Versionsableitung

Erneut beschreiben hier die Werte `total`, `load` und `save` die Gesamtlaufzeit der Funktionalität Versionsableitung beziehungsweise das Laden und Speichern der Modell-Datei. Der Wert `derive`

bezeichnet die reine Rechenzeit für das Durchführen der Versionsableitung. Auch hier sind alle Laufzeiten in Millisekunden angegeben.

Insgesamt ergibt sich in dieser Abbildung ein ähnliches Bild wie bei den Laufzeiten für die Transformation aus Abbildung 6.2. Allerdings ist die Gesamtlaufzeit mit knapp unter einer Sekunde für die Fallstudien Scheibenwischanlage, Minenpumpanlage und Verkaufsautomat ein wenig geringer als bei der Transformation, während die Laufzeit für das Body Comfort System mit knapp zweieinhalb Sekunden höher liegt als zuvor. Somit erhöht sich vor allem die Differenz zwischen den Fallstudien mit einer geringen Modell-Größe und dem Body Comfort System mit einer hohen Modell-Größe.

Auch für diese Funktionalität macht das Laden des Modells mit 84,5-90,7% wieder den größten Teil der Laufzeit aus. Das Speichern des abgeleiteten Modells macht hierbei diesmal mit 3,7-5,1% den geringsten Anteil der Gesamtlaufzeit aus. Auf einen ähnlich geringen Wert kommt die reine Rechenzeit für die Ableitung. Dieser liegt für die Fallstudien Scheibenwischanlage, Minenpumpanlage und Verkaufsautomat bei 5,1-6,2%, während der relative Zeitanteil mit 11,9% für das Body Comfort System schon doppelt so hoch liegt. Der absolute Zeitanteil in Millisekunden ist für das Body Comfort System sogar etwa fünf mal so hoch wie für die Fallstudien mit kleineren Modellen. Dies könnte an der vier bis fünf mal größeren Anzahl der Elemente liegen, über die während der Ableitung iteriert werden muss. Außerdem zeigt auch die Ableitungszeit für die Minenpumpanlage im Vergleich zu den Zeiten von Scheibenwischanlage und Verkaufsautomat einen verhältnismäßig hohen Ausschlag. Dies widerspricht einer Korrelation mit der Modell-Größe des Eingabe-Modells. Aus diesem Grund wird vermutet, dass sich die erhöhten Laufzeiten an dieser Stelle aufgrund einer höheren Komplexität der Modelle durch eine höhere Tiefe und mehr parallele Regionen ergeben.

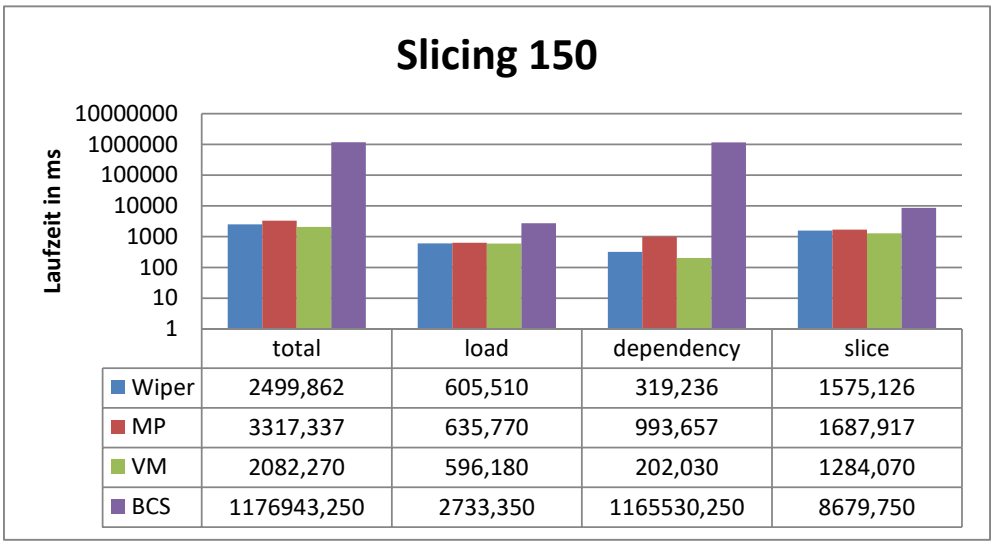
Version	1	2	3	4	5	6	7
Wiper	43,240	47,440	54,770	54,400	55,270		
MP	57,640	60,840	64,540				
VM	46,140	45,940	44,340	48,240	49,370	53,540	62,840
BCS	225,300	256,740	282,800	343,100			

Abbildung 6.4.: Zeitverhalten Versionsableitung für unterschiedliche Versionen

In der Tabelle in Abbildung 6.4 sind die Durchschnittswerte für die Testläufe der Ableitung der unterschiedlichen Versionen der Fallstudien in Millisekunden abgebildet. Die Laufzeiten zeigen eine steigende Tendenz im Verlauf der Versionen. Dies kann darauf zurückgeführt werden, dass die Evolutionsszenarien [29] für die Fallstudien in allen bis auf einen Fall eine Erhöhung der Komplexität des 175%-Modells durch das Hinzufügen von Funktionalität zur Folge haben. Lediglich in Version 3 des Verkaufsautomaten wird Funktionalität entfernt. Dies könnte die minimal kürzere Laufzeit bei der Ableitung dieser Version erklären.

150%-Slicing

In Abbildung 6.5 ist das Zeitverhalten für das Slicing von 150%-Modellen dargestellt. Auch hier wurde zunächst über die Testläufe mit der selben Version als Kriterium gemittelt, um dann den Durchschnitt aus diesen Durchschnittswerten zu bilden. Für diese Funktionalität sind wieder die Werte `total` und `load` für die Gesamtlaufzeit der Funktionalität und das Laden des Modells dargestellt.



	total	load	dependency	slice
Wiper	2499,862	605,510	319,236	1575,126
MP	3317,337	635,770	993,657	1687,917
VM	2082,270	596,180	202,030	1284,070
BCS	1176943,250	2733,350	1165530,250	8679,750

Abbildung 6.5.: Zeitverhalten 150%-Slicing

Zusätzlich finden sich hierbei die Werte `dependency` für die Erstellung eines Abhängigkeitsgraphen und `slice` für die Rechenzeit des reinen Slice-Vorgangs. Der Wert für `save` entfällt bei dieser Funktionalität, da es sich bei einem Slice nicht um ein Modell, sondern lediglich um eine Menge von Modellelementen handelt, die folglich nicht gespeichert wird. Die Laufzeiten sind wieder in Millisekunden angegeben, allerdings muss diesmal beachtet werden, dass die Skala logarithmisch skaliert ist. Die Unterschiede zwischen den Werten sind mit steigender Höhe folglich größer als sie auf der Abbildung wirken.

Für diese Funktionalität lässt sich nicht auf Anhieb erkennen, welche Phase des Slicing-Prozesses hauptsächlich für das Zustandekommen der Gesamtlaufzeit verantwortlich ist. Für einen einfacheren Vergleich sind die Laufzeiten für die Fallstudien Scheibenwischanlage, Minenpumpanlage und Verkaufsautomat in Abbildung 6.6 noch einmal ohne die Werte des Body Comfort Systems anhand einer linearen Skalierung visualisiert. Für diese drei Fallstudien macht offensichtlich der reine Slicing-Vorgang einen wesentlichen Anteil der Gesamtlaufzeit aus. Das Laden des 150%-Modells aus der XMI-Datei zeigt für alle drei Fallstudien einen relativ ähnlichen Wert, während das Erstellen des Abhängigkeitsgraphen sehr großen Schwankungen unterliegt. Letzteres nimmt für die Minenpumpanlage drei bis vier mal so viel Zeit wie für die Scheibenwischanlage oder den Verkaufsautomaten in Anspruch. Dieses Verhalten zeigt eine Korrelation mit der Modell-Größe der 150%-Modelle, welche für die Scheibenwischanlage durchschnittlich 149 Elemente, für den Verkaufsautomaten durchschnittlich 158 Elemente und für die Minenpumpanlage durchschnittlich 220 Elemente beträgt. Allerdings zeigt der Verkaufsautomat für das Erstellen des Abhängigkeitsgraphen und des Slices noch einmal niedrigere Laufzeiten als die Scheibenwischanlage, obwohl der Verkaufsautomat minimal größere 150%-Modelle besitzen. Dies deutet eher auf einen Zusammenhang mit der Modell-Komplexität hin. Da die Modell-Tiefe des Verkaufsautomaten geringer ist als die der Scheibenwischanlage, während die maximale Anzahl an parallelen Regionen bei der Scheibenwischanlage höher ist als beim Verkaufsautomaten, liegt die Vermutung nahe, dass die Parallelität der Regionen hierbei der entscheidende Faktor ist. Diese Vermutung ergibt in Anbe-

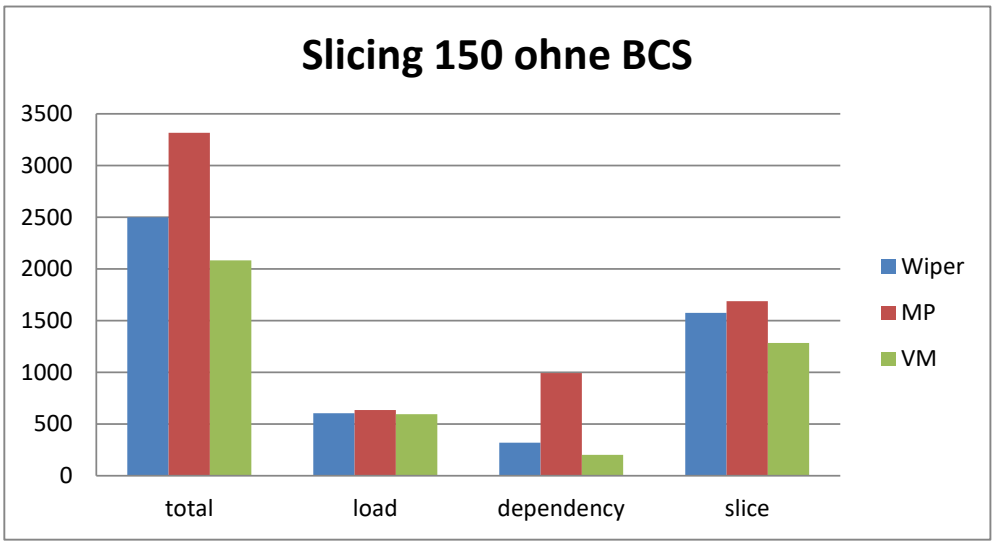


Abbildung 6.6.: Zeitverhalten 150%-Slicing ohne Body Comfort System

Version	1	2	3	4	5	6	7
Wiper	114,800	225,100	268,400	451,340	536,540		
VM	65,340	68,170	63,000	154,600	221,900	333,970	507,200

Abbildung 6.7.: Laufzeit reine Slicing-Phase für verschiedene Versionen

tracht der aufgrund der parallelen Regionen steigenden Anzahl von parallelen Abhängigkeiten und Verfeinerungskontrollabhängigkeiten (s. Kapitel 4) durchaus Sinn.

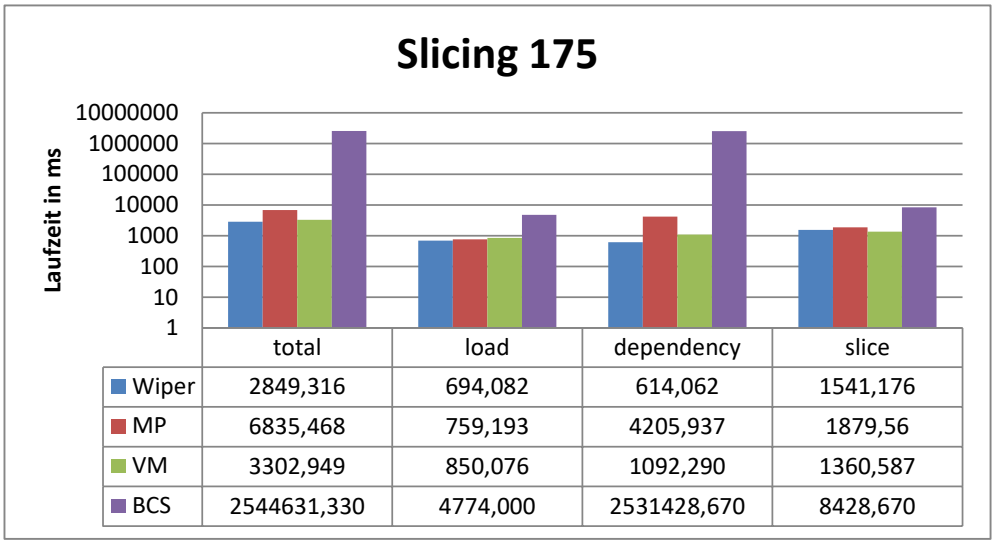
Diese Vermutung deckt sich mit den Zahlen aus Abbildung 6.7. Diese Tabelle zeigt die durchschnittlichen Laufzeiten zur Erstellung eines Abhängigkeitsgraphen beim 150%-Slicing für die unterschiedlichen Versionen von Scheibenwischanlage und Verkaufsautomat. Auffällig ist an dieser Stelle, dass für den Verkaufsautomaten von Version 3 auf Version 4 ein wesentlicher Anstieg der Laufzeit des 150%-Slicing erfolgt, da die Laufzeit sich mehr als verdoppelt. Bis einschließlich Version 3 befindet sich unter der Wurzel-Region des Verkaufsautomaten lediglich eine einzige Region. In Version 4 wird auf Level 2 eine zweite, parallele Region hinzugefügt und eine weitere in Version 5. Die Scheibenwischanlage besitzt hingegen bereits in Version 1 unter der Wurzel-Region zwei parallele Regionen und erhält in den Version 2, 4 und 5 zusätzliche, parallele Versionen auf Level 2. Des Weiteren ist das 150%-Modell der Scheibenwischanlage für Version 1 mit 82 Elementen noch einmal kleiner als das 150%-Modell des Verkaufsautomaten für Version 1 mit 132 Elementen. Folglich könnte die Tatsache, dass die Erstellung eines Abhängigkeitsgraphen für das kleinere Modell länger dauert, an dieser Stelle also tatsächlich auf die höhere Anzahl an parallelen Regionen zurückgeführt werden. Der Anstieg der Laufzeit beim Erstellen des Abhängigkeitsgraphen des Verkaufsautomaten für die Versionen 6 und 7, obwohl keine zusätzliche parallele Region hinzugefügt wird, ist vermutlich auf die extreme Erweiterung der Funktionalität in diesen Versionen zurückzuführen. Die Größe der Modell-Datei für Version 7 beträgt hier bereits 261 Elemente, während sie für Version 5 der Scheibenwischanlage lediglich 200 Elemente beträgt. Da die Laufzeit für Version 5 der Scheibenwischanlage dennoch höher ist als für Version 7 des Verkaufsautomaten, bestätigt dies

wiederum erneut die Vermutung, dass die Anzahl der parallelen Regionen hierbei den Ausschlag für die Laufzeit gibt.

Diese Beobachtungen decken sich außerdem mit den Werten für die Minenpumpanlage und das Body Comfort System aus Abbildung 6.5. Dabei ist die Zeit zur Erstellung des Abhängigkeitsgraphen für die Minenpumpanlage schon wesentlich höher als die gleiche Laufzeit bei Scheibenwischanlage und Verkaufsautomat. Für das Body Comfort System, welches die größte Anzahl an parallelen Regionen aufweist, steigt die Laufzeit für diese Phase exorbitant und macht mit 19,43 von insgesamt 19,62 Minuten 99% der Gesamtlaufzeit aus. Dies zeigt wiederum einen Unterschied zur Minenpumpanlage, bei der mit 50,9% (1,688 Sekunden) die reine Slicing-Phase den größten Teil der Gesamtlaufzeit ausmacht. Hier dauert die Slicing-Phase also fast doppelt so lang wie das Erstellen des Abhängigkeitsgraphen. Es liegt folglich die Vermutung nahe, dass die Laufzeit für die Erstellung eines Abhängigkeitsgraphen die Laufzeit der Slicing-Phase ab einem bestimmten Grenzwert an Parallelität im Modell übersteigt. Für kleine Modelle mit geringer Parallelität ist demnach die Slicing-Phase der wesentliche Laufzeit-Faktor, während die Laufzeit für das Erstellen von Abhängigkeitsgraphen mit Modell-Größe und steigender Parallelität stark zunimmt.

175%-Slicing

Das Zeitverhalten für das Slicing von 175%-Modellen ist in Abbildung 6.8 dargestellt. Dabei wurde zunächst wieder der Durchschnitt für alle Testläufe mit der gleichen Version als Kriterium gebildet, um aus diesen Ergebnissen dann den Gesamtdurchschnitt zu bilden. Die unterschiedlichen Phasen des Slicings, die hier betrachtet werden, sind die gleichen Phasen wie beim 150%-Slicing. Die Laufzeit ist wieder in Millisekunden auf einer logarithmischen Skala dargestellt.



	total	load	dependency	slice
Wiper	2849,316	694,082	614,062	1541,176
MP	6835,468	759,193	4205,937	1879,56
VM	3302,949	850,076	1092,290	1360,587
BCS	2544631,330	4774,000	2531428,670	8428,670

Abbildung 6.8.: Zeitverhalten 175%-Slicing

Hierbei zeigt sich eine ähnliche, relative Verteilung der Werte wie zuvor beim 150%-Slicing. Bemerkenswert ist an dieser Stelle vor allem, dass die reine Slicing-Phase ähnliche, absolute Laufzeiten aufweist wie beim 150%-Slicing in Abbildung 6.5, während sich die absoluten Laufzeiten für das Erstellen eines Abhängigkeitsgraphen verdoppelt bis verfünffacht haben. Daraus lässt sich schließen, dass die Laufzeit der reinen Slicing-Phase weitestgehend unabhängig von Modell-Größe oder

-Komplexität durch Parallelität ist, während die Erstellung eines Abhängigkeitsgraphen, wie bereits vermutet, wesentlich davon abhängt. Dieser Schluss kann gezogen werden, da nicht jedes 150%-Modell sämtliche parallelen Regionen enthält und die 175%-Modelle somit komplexer sind als die 150%-Modelle.

Nicht rational erklärt werden, kann an dieser Stelle der extreme Anstieg der Laufzeit für die Erstellung des Abhängigkeitsgraphen des Verkaufsautomaten. Dieser Wert hat sich im Vergleich zum 150%-Slicing um das Fünffache gesteigert, während sich der gleiche Wert für die Scheibenwischanlage, welche eine ähnliche, leicht höhere Komplexität aufweist, nur verdoppelt hat. Eventuell liegen an dieser Stelle Messfehler durch eine gleichzeitige, erhöhte Nutzung der CPU durch andere Anwendungen vor.

Die Steigerung der Gesamtlaufzeit des 175%-Slicings im Vergleich zur Gesamtlaufzeit des 150%-Slicings ist an dieser Stelle somit hauptsächlich auf die Steigerung der Laufzeit für die Erstellung des Abhängigkeitsgraphen zurückzuführen. Die Laufzeiten für das Laden der Modelle und das Erstellen des Slices ändern sich nur bedingt.

Vergleich 175%-Slicing und Kombination Versionsableitung - 150%-Slicing

In diesem Abschnitt sollen die Laufzeiten einer kombinierten Anwendung von Versionsableitung und 150%-Slicing mit den Laufzeiten des 175%-Slicings verglichen werden. Dazu zeigt Abbildung 6.9 für die verschiedenen Fallstudien die Gesamtlaufzeit des 175%-Slicings im Vergleich mit den addierten Gesamtlaufzeiten des 150%-Slicings und der Versionsableitung (*Derivation*). Auch hierbei ist die Laufzeit in Millisekunden wieder logarithmisch skaliert.

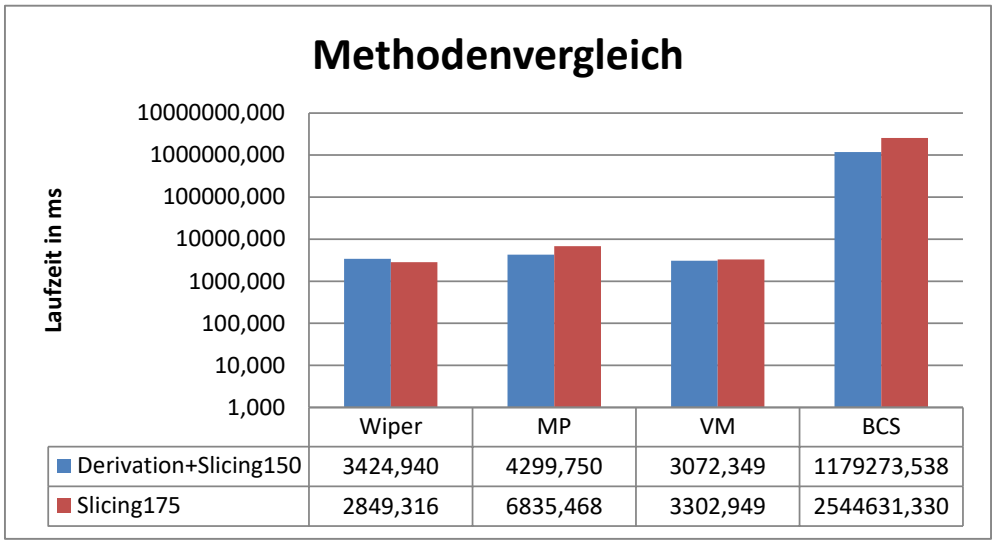


Abbildung 6.9.: Zeitverhalten Vergleich 175%-Slicing und Kombination Versionsableitung - 150%-Slicing

Die Abbildung zeigt eindeutig, dass das 175%-Slicing in den meisten Fällen höhere Laufzeiten erzielt als die Kombination aus den beiden anderen Methoden. Lediglich für die Scheibenwischanlage erzielt das 175%-Slicing eine kürzere Laufzeit. Dies ist darauf zurückzuführen, dass die Gesamtlaufzeit für das 150%-Slicing der Scheibenwischanlage nicht wesentlich geringer ist als die Gesamtlaufzeit für das 175%-Slicing. Durch die zusätzliche Laufzeit für die Versionsableitung über-

steigt das Ergebnis der Kombination somit das Ergebnis des 175%-Slicings. Im Falle der anderen drei Fallstudien ist die Laufzeit des 175%-Slicing rund doppelt so lang wie die Laufzeit des 150%-Slicings. Da die Laufzeit der Versionsableitung 47% (Verkaufsautomat), 29% (Minenpumpanlage) beziehungsweise 0,1% (Body Comfort System) der Laufzeit des 150%-Slicings benötigt, reichen die kombinierten Laufzeiten nicht aus um die Laufzeit des 175%-Slicings zu übersteigen.

Zusätzlich muss in Betracht gezogen werden, dass die Gesamtlaufzeiten für die Kombination einfach nur addiert wurden. Das bedeutet, dass das 175%-Modell für die Versionsableitung aus einer Datei geladen und dann als 150%-Modell in eine Datei gespeichert wurde. Diese Datei mit dem 150%-Modell wurde wiederum für das 150%-Slicing geladen. Prinzipiell könnten beide Funktionalitäten allerdings ohne ein Zwischenspeichern des 150%-Modells gekoppelt werden. In diesem Fall würde die Speicherzeit der Versionsableitung und die Ladezeit des 150%-Slicings wegfallen. Die Speicherzeit ist aufgrund ihres geringen Wertes zwar unwesentlich, die Ladezeit macht hingegen für die Scheibenwischanlage und den Verkaufsautomaten rund ein Viertel der Gesamtlaufzeit des 150%-Slicings aus. Abzüglich dieser Laufzeit wäre somit auch die Laufzeit der Kombination für die Scheibenwischanlage geringer als die Laufzeit des 175%-Slicings.

Folglich bleibt festzuhalten, dass 175%-Slicing aufgrund der stark steigenden Laufzeit für die Erzeugung des Abhängigkeitsgraphen lediglich für sehr kleine Modelle mit einer geringen Komplexität lohnenswert ist. Für größere und komplexere Modelle, die für Softwareproduktlinien in den meisten Fällen anzutreffen sind, sollte daher - zumindest bei einem Slicing-Kriterium mit nur einer Version - lieber auf die Kombination aus Versionsableitung und 150%-Slicing zurückgegriffen werden. Für ein Slicing mit einem Kriterium, das mehr als eine Version enthält, ist das 175%-Slicing hingegen alternativlos, da ein solcher Slice mit 150%-Slicing nicht möglich ist.

Zusammenfassend ist für die Evaluation der Laufzeiten der verschiedenen Funktionalitäten somit zu sagen, dass die durchschnittliche Laufzeit bei Transformation und Versionsableitung zwischen einer und drei Sekunden liegt, wobei die Gesamtlaufzeit hauptsächlich aus der Ladezeit des Modells besteht. Beim Slicing beträgt die Gesamtlaufzeit hingegen zwischen zwei Sekunden und 45 Minuten und hängt dabei wesentlich vom variablen Faktor der Laufzeit zur Erstellung des Abhängigkeitsgraphen ab. Diese Zeit steigt sehr stark an, je komplexer das Modell aufgebaut ist, da dann folglich mehr Abhängigkeiten bestehen können. Entsprechend müsste an diesen Stellen angesetzt werden, wenn eine Laufzeitverbesserung angestrebt wird. Beim Slicing eines 175%-Modells für nur eine Version benötigt in den meisten Fällen das 175%-Slicing mehr Laufzeit als die Kombination aus Versionsableitung und 150%-Slicing. Daher wird empfohlen in diesem Fall die letztere Methode anzuwenden.

Bei den Testdurchläufen und Messungen wurde versucht, störende Einflüsse soweit wie möglich zu vermeiden, um aussagekräftige Ergebnisse zu erhalten. Solche Störfaktoren können beispielsweise Laufzeitunterschiede durch unterschiedlich leistungsstarke Testgeräte oder, wie beim 175%-Slicing bereits angedeutet, Prozessunterbrechungen durch höher priorisierte Prozesse sein. Um Unterschiede zwischen Geräten mit unterschiedlichen Leistungen auszuschließen, wurden sämtliche Testläufe auf dem selben Gerät durchgeführt. Zusätzlich wurden zwischen Beginn und Ende der Testdurchläufe keine Änderungen an den Einstellungen oder Installationen des Geräts, auf dem die Testdurchläufe durchgeführt wurden, und der verwendeten Software vorgenommen, um für jeden Durchlauf gleiche Ausgangsbedingungen zu schaffen. Zur Vermeidung von Prozessunterbrechungen durch anderen Prozesse wurde die aktive Nutzung des Testgeräts während der Durchläufe auf

ein Minimum beschränkt. Dennoch kann an dieser Stelle nicht garantiert werden, dass keinerlei Hintergrundprozesse Einfluss auf die Laufzeiten genommen haben. Aus diesem Grund wurden sämtliche Testläufe mehrmals durchgeführt, um einzelne Ausschläge in der Laufzeit, die wegen Verzögerungen durch Hintergrundprozesse entstanden sein könnten, ausgleichen zu können.

Im folgenden Kapitel werden nun die Gesamtergebnisse dieser Arbeit noch einmal zusammengefasst. Des Weiteren wird ein Ausblick für zukünftige Arbeiten gegeben.

7 Zusammenfassung und Fazit

Das Ziel der Arbeit war zum einen die Konzeptionierung der 175%-Modellierung, einer Modellierungsmethode für Evolution in annotativen Variabilitätsmodellen. Hiermit wurde der Zweck verfolgt, auch für annotative Modellierungsverfahren eine Methode zu entwickeln, mit der Variabilität und Evolution auf die gleiche Weise dargestellt werden können. Auf Basis dieser Modellierungstechnik sollte dann das bedingte Modell-Slicing, das bereits für 150%-Modelle besteht, für 175%-Modelle erweitert werden, um auch die komplexen 175%-Modelle auf für ein bestimmtes Kriterium relevante Inhalte reduzieren und somit gezieltere Analysen durchführen zu können. Zum anderen sollte ein Algorithmus zur Umwandlung von Higher-Order Delta-Modellen in 175%-Modelle entwickelt werden. Mit Hilfe dieses Algorithmus sollen beide Ansätze ohne Mehraufwand parallel verwendet werden können. Durch Verwendung beider Ansätze sind Vergleiche oder das Ausgleichen von Vor- und Nachteilen der jeweiligen Modellierungstechniken möglich. Darüber hinaus sollten sowohl die 175%-Modelle samt einer Möglichkeit zur Ableitung von einzelnen Versionen von 150%-Modellen aus den 175%-Modellen als auch der Transformationsalgorithmus und das 175%-Slicing implementiert werden.

Zu diesem Zweck wurden zu Beginn der Arbeit die wesentlichen Grundlagen für diese Thematik erarbeitet. Dazu wurde zunächst sowohl das Konzept der Softwareproduktlinien vorgestellt, als auch verschiedene Mechanismen diese zu modellieren. Zu diesen zählen etwa Techniken zur Modellierung der Variabilität auf Ebene der kompletten Produktlinie, wie beispielsweise Feature-Modelle, oder auch Methoden zur Modellierung auf Artefaktebene. Hierbei wurden verschiedene annotative, transformationale und kompositionale Ansätze vorgestellt. Der wesentliche Fokus wurde jedoch auf 150%-Modelle beziehungsweise Feature-annotierte State Machines gelegt, da diese den Grundstein für die 175%-Modellierung bilden. Zusätzlich wurde ebenfalls Delta-Modellierung näher betracht, da die Higher-Order Delta-Modellierung, welche für den Transformationsalgorithmus benötigt wurde, darauf aufbaut. Des Weiteren wurden als Basis für das 175%-Slicing sowohl normales Slicing als auch Modell-Slicing, und hierbei besonders 150%-Slicing, erläutert. Danach folgten theoretische Aspekte zur Evolution von Softwareproduktlinien und einige Möglichkeiten, diese zu modellieren. Dafür wurden verschiedene Arten, Evolution für Feature-Modelle darzustellen, beschrieben, sowie die Higher-Order Delta-Modellierung als Technik zu Modellierung von Evolution auf Artefaktlevel.

Nachdem so die Grundlagen für das Verständnis der weiteren Arbeit erarbeitet wurden, erfolgte die Erweiterung des Konzeptes der 150%-Modellierung auf 175%-Modellierung. Während bei der 150%-Modellierung jedem Element eine Feature-Bedingung in Form einer aussagenlogischen Formel über Features zugeordnet wird, wird bei der 175%-Modellierung jedem Element nun eine Menge von Tupeln zugeordnet. Diese Tupel enthalten jeweils eine einzigartige Version aus der Versionsmenge der Evolutionshistorie der Produktlinie sowie die Feature-Bedingung des Elements aus dem 150%-Modell der entsprechenden Version. Das 150%-Slicing wurde auf ein 175%-Slicing erweitert, indem im Slicing-Kriterium, auf dessen Basis der Slice erstellt wird, nun zusätzlich auch eine Menge von Versionen angegeben werden kann.

Anschließend wurde der Algorithmus konzeptioniert, der es erlaubt, Higher-Order Delta-Modelle in diese neuen 175%-Modelle umzuwandeln. Dazu wird jedes Element, das mindestens einmal im Higher-Order Delta-Modell vorkommt, in das 175%-Modell eingefügt und nach Regeln, die sich auf Basis der Kombination aus Delta-Operation und Element-Operation bilden, annotiert. Nach der Definition des Algorithmus wurde außerdem die korrekte Funktion des Algorithmus durch mathematische Induktion bewiesen. Daraus ergibt sich, dass der Algorithmus in jedem Fall ein 175%-Modell erzeugt, das äquivalent zum eingegebenen Higher-Order Delta-Modell ist. Das bedeutet beide Modelle erzeugen für eine gegebene Feature-Konfiguration und eine gegebene Version das gleiche Produktmodell.

Sowohl die 175%-Modellierung, der Transformationsalgorithmus und das 175%-Slicing wurden in unterschiedlichen Eclipse-Plug-ins implementiert, um eine einfache Einbettung und Erweiterbarkeit zu garantieren. Zusätzlich wurde eine Möglichkeit implementiert, einzelne Versionen von 150%-Modellen aus einem 175%-Modell abzuleiten.

Des Weiteren erfolgte die Dokumentation dieser vier Funktionalitäten. Hierzu wurde zunächst eine bestehende Implementierung von 150%-Modellen, Higher-Order Delta-Modellen und 150%-Slicing vorgestellt, auf der die neue Implementierung aufbauen sollte. Danach wurde die neue beziehungsweise erweiterte Implementierung beschrieben, indem der Aufbau, Arbeitsablauf und die Funktionsweise sowie die Möglichkeiten zur Anwendung erläutert wurden.

Abschließend wurde im Rahmen einer Evaluation das Zeitverhalten der implementierten Funktionalitäten unter Verwendung von vier verschiedenen Fallstudien ausgewertet. In diesem Zusammenhang erfolgte außerdem ein Vergleich der Laufzeiten des 175%-Slicings mit den Laufzeiten einer Kombination aus Versionsableitung und 150%-Slicing.

Im Folgenden werden Erkenntnisse vorgestellt, die während der Erstellung der Arbeit gewonnen werden konnten.

Erkenntnisse

Zunächst einmal kann festgehalten werden, dass mit dem Konzept der 175%-Modellierung von State Machines ein annotativer Ansatz geschaffen wurde, der sowohl Variabilität als auch Evolution durch das gleiche Verfahren repräsentiert. Auf diese Weise ist ein einheitlicher Umgang mit diesen Informationen möglich, der außerdem das Modell-Verständnis und etwaige Modell-Analysen erleichtert. Mit dem Transformationsalgorithmus wurde darüber hinaus eine schnelle, einfach anwendbare Möglichkeit zur Umwandlung von Higher-Order Delta-Modellen entwickelt. Hierbei muss lediglich das Eingabe-Modell ausgewählt werden, die Transformation läuft automatisch ab, sodass keinerlei Aufwand für den Anwender entsteht. Ebenso wurde das bestehende Slicing für 175%-Modelle erweitert, sodass auch auf diesen Modellen Analysen durchgeführt werden können.

Durch die parallele Verwendung von 175%-Modellen und Higher-Order Delta-Modellen bei der Evaluation der Funktionalitäten sind außerdem Vor- und Nachteile bei diesen Ansätzen zutage getreten. So werden bei der 175%-Modellierung zwar sämtliche Elemente in einer State Machine betrachtet, wodurch diese sehr groß wird, dennoch erwies sich diese Art der Darstellung trotz der Größe als wesentlich übersichtlicher als die Darstellung durch ein Higher-Order Delta-Modell.

Dies liegt vor allem daran, dass die Struktur einer State Machine durch die 175%-Modelle beibehalten wird. Folglich ist sofort ersichtlich, welches Element sich an welcher Stelle im Modell befindet beziehungweise wie es dort eingebettet ist und welchen Bezug es zu anderen Modellelementen

besitzt. Bei Higher-Order Delta-Modellen ist dies nicht der Fall. Durch die Strukturierung über Additions, Removals und Modifications wird erst nach genauerem Hinsehen deutlich, an welcher Stelle ein Element hinzugefügt oder von wo im Modell es entfernt werden soll beziehungsweise welche Verbindungen zu anderen Elementen bestehen.

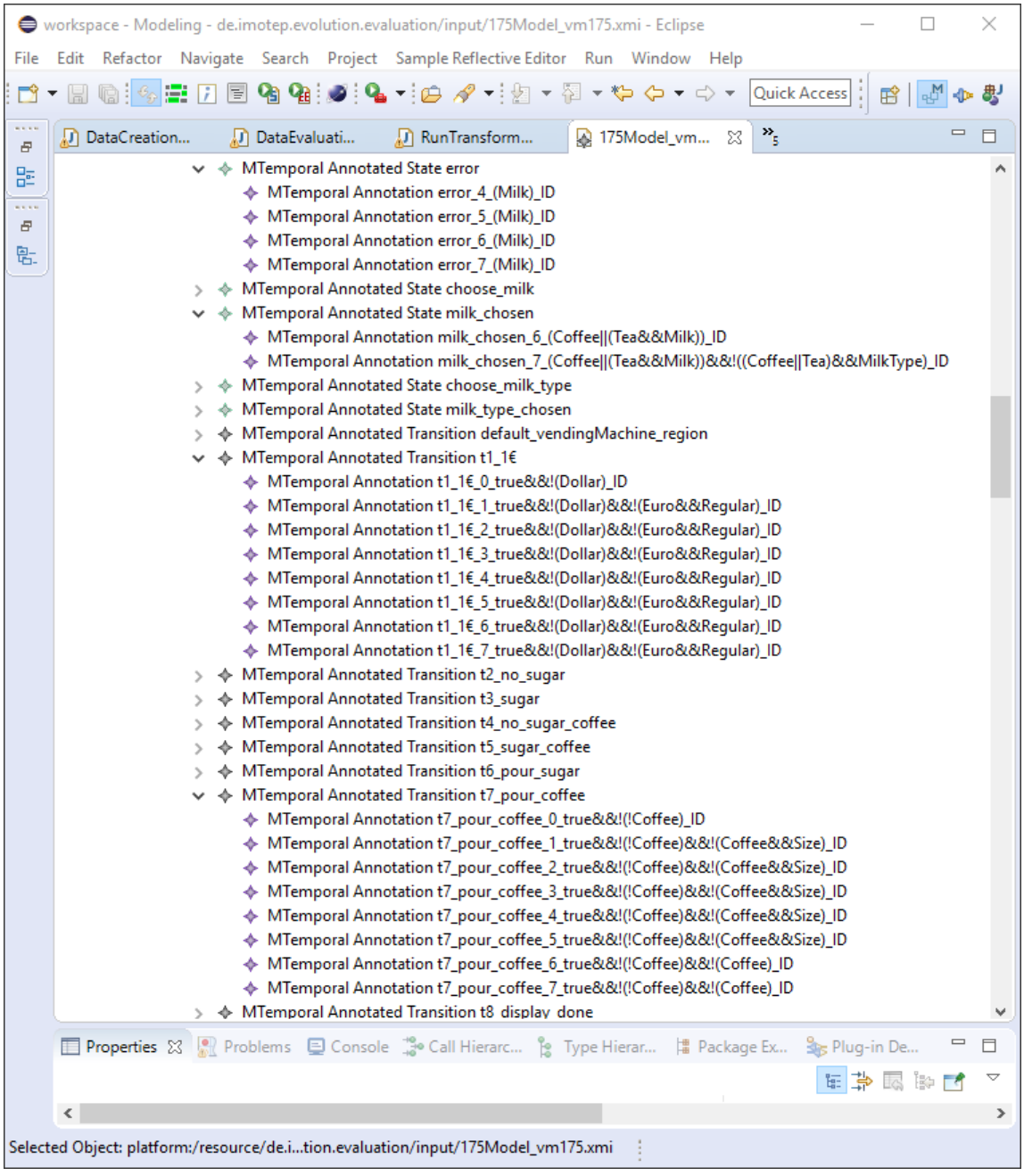


Abbildung 7.1.: Darstellung der Annotationen von 175%-Elementen im Editor

Gleiches gilt für die Darstellung von Variabilität und Evolution. Für 175%-Modelle ist ohne Aufwand erkennbar, unter welchen Bedingungen ein Element in einem Produktmodell enthalten ist. Hierfür ist ein Beispiel in Abbildung 7.1 dargestellt. Die Elemente besitzen für jede Version, in der sie enthalten sind, ein eigenes Annotationsobjekt. So wird sofort deutlich, dass der Zustand `error`

in den Versionen 4 bis 7 enthalten ist, während der Zustand `milk_chosen` nur in den Versionen 6 und 7 und die Transition `t1_1€` sogar in allen acht Versionen vorkommt. Auch die Bedingungen für die Existenz eines Elements in einem Produktmodell innerhalb einer Version sind leichter zu erkennen. In Version 6 ist der Zustand `milk_chosen` etwa nur in einem Produktmodell enthalten, wenn entweder das Feature `Coffee` oder die Features `Tea` und `Milk` selektiert sind. In Version 7 ist der Zustand jedoch nur enthalten, wenn zuerst wieder die Features `Coffee` oder `Tea` und `Milk` ausgewählt sind, gleichzeitig darf diesmal jedoch nicht das Feature `MilkType` selektiert sein.

Bei Higher-Order Delta-Modellen sind diese Zusammenhänge nicht direkt ersichtlich. Wird ein Element an einer Stelle hinzugefügt oder ist es von Beginn an enthalten, muss das gesamte Modell durchgesucht, um alle Delta- und Element-Operationen zu finden, die auf dieses Element Einfluss nehmen. Nur aus der Gesamtheit dieser Operationen kann festgestellt werden, in welchen Versionen und unter welchen Bedingungen ein Element in einem Produkmodell vorhanden ist.

Folglich bieten 175%-Modelle einen besseren Überblick als Higher-Order Delta-Modelle. Wenn mit Higher-Order Delta-Modellen gearbeitet wird, ist eine ergänzende Verwendung von 175%-Modellen daher durchaus sinnvoll. Eine Anwendung des Transformationsalgorithmus spart an dieser Stelle wertvolle Zeit bei der Modellerstellung.

Gleichwohl sind auch bei der Verwendung des Algorithmus verschiedene Auffälligkeiten zutage getreten. Zunächst einmal wurde der Algorithmus unter der Prämisse erstellt, dass ein korrektes und wohlgeformtes Higher-Order Delta-Modell als Eingabe verwendet wird. Der Algorithmus ist daher nicht in der Lage, bei der Transformation Fehler auszugleichen, die bereits zuvor im Higher-Order Delta-Modell bestanden haben, wie beispielsweise doppelt vergebene IDs.

Sofern doppelte IDs Elemente aus unterschiedlichen Klassen betreffen, erstellt der Algorithmus strukturell auf jeden Fall ein richtiges Modell. Allerdings besitzt auch dieses hinterher doppelte IDs. Sollten die doppelten IDs allerdings Elemente der gleichen Klasse betreffen, ist der Algorithmus nicht in der Lage, die korrekte Struktur des Modells aufrechtzuerhalten. Dies liegt daran, dass bei der Überprüfung, ob ein Element bereits im 175%-Modell enthalten ist oder neu hinzugefügt werden muss, die ID als Referenzwert genommen wird, da diese in einem korrekten Eingabe-Modell einzigartig sein sollte. Wenn dann in der entsprechenden Klassenliste nach dem Element gesucht wird, wird folglich nur die ID verglichen. Wird dabei die gleiche ID entdeckt, geht der Algorithmus davon aus, dass es sich um dasselbe Element handelt. Bei diesem wird dann lediglich die Annotation der neuen Operation angepasst, statt ein vollkommen neues Element hinzuzufügen. Bei Elementen unterschiedlicher Klassen tritt dieses Problem nicht auf, da nur die Klassenliste des Elementes auf ein Element mit gleicher ID geprüft wird. Folglich wird das Element neu erstellt, wenn in der eigenen Klassenliste kein Element mit der gleichen ID vorhanden ist, auch wenn in einer anderen Klassenliste ein Element die gleiche ID haben sollte.

Des Weiteren muss beim ersten Hinzufügen von Elementen im Higher-Order Delta-Modell auf den Aufbau der Deltas geachtet werden, damit Elemente, die von anderen Elementen referenziert werden, zuerst erstellt werden können. Befinden sich alle betroffenen Element in einer einzigen Addition übernimmt der Algorithmus das Beachten der richtigen Reihenfolge bei der Erstellung der Elemente. Sind die Elemente allerdings in unterschiedlichen Additions des gleichen Deltas untergebracht, muss darauf geachtet werden, dass die Additions entsprechend der Referenzierungsreihenfolge (Attribute → Events → Actions/Guards → Regionen → Zustände → Transitionen) richtig angeordnet sind.

Dies liegt daran, dass die Additions in der Reihenfolge abgearbeitet werden, in der sie im Delta vorkommen. Die Elemente der ersten Addition in der Liste werden also vor den Elementen der zweiten Addition erstellt. Befindet sich beispielweise in der ersten Addition eine Transition mit Referenz auf einen Guard, der erst in der zweiten Addition erstellt wird, kann dieser Guard für diese Transition nicht referenziert werden, da er zum Zeitpunkt der Erstellung der Transition noch nicht existiert. Werden einzelne Elemente also in unterschiedlichen Additions verpackt, muss hierbei auf die richtige Anordnung der Additions im Delta geachtet werden.

Zusammenfassend hängt die Ausgabe eines korrekten 175%-Modells somit von einem korrekten Aufbau des eingegebenen Higher-Order Delta-Modells ab. Aufgrund der Vorgabe, dass das eingegebene Higher-Order Delta-Modell korrekt, konfliktfrei und wohlgeformt sein muss, ist dieser Befund in Ordnung.

Darüber hinaus ist auffällig, dass die resultierenden Feature-Bedingungen in den Annotationen nicht unbedingt einfach leserlich formatiert sind. In Abbildung 7.1 besitzt die Annotation von Transition `t7_pour_coffee` für Version 6 etwa die Feature-Bedingung $true \wedge \neg(\neg Coffee) \wedge \neg(Coffee)$. Bei näherer Betrachtung beläuft sich diese Feature-Bedingung also auf $false$, da die Transition ab Version 6 nicht mehr benutzt wird. Je nach Anzahl der Deltas, die pro Version auf einem Element operieren, und der Komplexität ihrer Anwendungsbedingung, werden auch die Feature-Bedingungen entsprechend komplex. Eine besser lesbare Darstellung der Bedingungen, etwa durch ein Vereinfachen der aussagenlogischen Formeln, wäre an dieser Stelle wünschenswert. Hierbei können die Annotationen jedoch nicht einfach vereinfacht werden. Werden Deltas aus dem Higher-Order Delta-Modell entfernt, werden aus den Feature-Bedingungen der Annotationen die Terme entfernt, die der Anwendungsbedingung des Deltas entsprechen. Bei einer Vereinfachung der Formeln in den Annotationen wäre der Algorithmus nicht mehr in der Lage zu identifizieren, welcher Teil der Formel entfernt werden müsste, um die gewünschte Änderung darzustellen. Folglich müsste eine Trennung der Annotationen, auf denen die Berechnung stattfindet, und der vereinfachten Annotationen, die dem Benutzer angezeigt werden sollen, erfolgen. Auf den Vergleich der Feature-Bedingung mit der Feature-Konfiguration des Slicing-Kriteriums beim Slicing hat der kompliziertere Aufbau der Feature-Bedingungen keinen Einfluss und stört folglich nicht.

Auch bei der Verwendung des Slicings sind Auffälligkeiten hervorgetreten. Wird beispielsweise eine partielle Feature-Konfiguration für das Slicing-Kriterium gewählt, findet bei Erstellen eines Slices noch kein automatisches Aussortieren von bestimmten Elementen aufgrund von Beziehungen zwischen den Features statt. Dies liegt daran, dass beim Slicing zurzeit noch kein Einbezug von Feature-Modellen stattfindet. Wird also eine partielle Feature-Konfiguration als Kriterium eingegeben, bei der ein Feature selektiert ist, das eine excludes-Beziehung zu einem anderen Feature besitzt, ohne dass über dieses andere Feature in der Konfiguration eine Aussage getroffen wird, dann sind auch Elemente im Slice enthalten, die für dieses andere Feature gelten. Diese Ergebnis ist natürlich nicht korrekt, da solche Elemente für die gewählte Feature-Konfiguration nicht im Produktmodell enthalten sein können. Dementsprechenden dürften diese Elemente auch nicht im Slice enthalten sein.

In diesem Zusammenhang ist außerdem bei der Evaluation des Zeitverhaltens aufgefallen, dass sich für partielle Feature-Konfigurationen als Slicing-Kriterium tendenziell längere Laufzeiten als für vollständige Feature-Konfigurationen ergaben. An dieser Stelle gilt, je detaillierter die Konfiguration, desto weniger Laufzeit benötigte das Slicing. Dies lässt sich darauf zurückführen, dass die

Menge der Elemente im Slice in den meisten Fällen durch eine vollständige Feature-Konfiguration im Vergleich zu einer partiellen Feature-Konfiguration eingeschränkt wird. Werden bestimmte Elemente aufgrund der Konfiguration nicht in den Slice gewählt, dann werden im nächsten Schritt deren abhängige Elemente im Abhängigkeitsgraphen nicht mehr überprüft. Folglich wird die Zeit eingespart, die bei einer Konfiguration, welche keine Aussage über dieses Element trifft, sodass das Element zum Slice hinzugefügt wird, zur Berechnung der abhängigen Elemente dieses Elements angefallen wäre. Da somit auch rekursiv alle weiteren Abhängigkeiten der abhängigen Elemente aus der Betrachtung fallen, wird also bei Eingabe einer vollständigen Konfiguration tendenziell weniger Zeit zur Berechnung des Slices benötigt.

Im folgenden Abschnitt sollen auf dieser Arbeit aufbauende Möglichkeiten für zukünftige Arbeiten vorgestellt werden.

Ausblick

In dieser Arbeit wurde ein Algorithmus zur Transformation von Higher-Order Delta-Modellen in 175%-Modelle vorgestellt. Eine Möglichkeit zur Transformation von 175%-Modellen in Higher-Order Delta-Modelle existiert jedoch nicht. Sollte bereits ein 175%-Modell vorliegen, müsste folglich doppelter Aufwand betrieben werden, wenn beide Modellformen verwendet werden sollen. In diesem Fall müsste das Higher-Order Delta-Modell dann noch zusätzlich per Hand erstellt werden. Aus diesem Grund wäre die Konzeptionierung eines Transformationsalgorithmus für die andere Richtung ein lohnenswerter Ansatz.

Für den bisherigen Transformationsalgorithmus könnte eine Erkennung von mehrfach verwendeten IDs implementiert werden. Dies würde für Objekte unterschiedlicher Klassen mit gleichen IDs funktionieren, wenn die verschiedenen Klassenlisten auf gleiche IDs überprüft werden. Elemente der gleichen Klasse mit gleicher ID könnten so jedoch nicht ausfindig gemacht werden. In diesem Zusammenhang ergibt sich die Überlegung eine bessere Methodik zum Vergleich von Elementen als über den Vergleich der ID zu entwickeln. Im Rahmen dieser Arbeit wurde an dieser Stelle ein Vergleich über die Werte der Attribute und Referenzen in Erwägung gezogen. Jedoch stellte hierbei der rekursive Vergleich der Referenzen durch die gleiche Vorgehensweise aufgrund des erhöhten Rechenaufwands ein Problem dar.

Eine weitere Möglichkeit ist die Implementierung der 175%-Modellierung für andere Modellformen wie etwa Klassen- oder Komponentendiagramme. Im Rahmen dieser Arbeit wurde das Konzept speziell für State Machines verwendet, es kann prinzipiell jedoch auf jede andere Art von Modell angewendet werden.

Überdies könnte an dieser Stelle außerdem eine Möglichkeit zur besseren Darstellung und Erstellung von 175%-Modellen, zum Beispiel in Form von grafischen Editoren, entwickelt werden. Diese bieten eine weniger abstrakte Repräsentation der Modelle als die durch das Eclipse Modeling Framework automatisch generierten Editoren und erleichtern dementsprechend das Modellverständnis oder auch das Auffinden von Fehlern im Modell - wie beispielsweise einen nicht gesetzten Zielzustand einer Transition oder einen nicht gekennzeichneten Startzustand einer Region.

Weiterhin ist die Evaluation von Anwendbarkeit und Nutzen der 175%-Modelle von Interesse. So könnte beispielsweise die Nutzbarkeit der 175%-Modellierung zur automatischen Testfallgenerierung oder für Familien-basierte Analysestrategien, und natürlich auch für das Modell-Slicing, geprüft werden. Hierbei stellt sich die Frage, ob die 175%-Modelle für diese Anwendungsgebie-

te wie vermutet eingesetzt werden können, und wenn ja, wie erfolgreich sie dabei sind. Auch das 175%-Slicing bietet an dieser Stelle Potenzial für eine solche Beurteilung.

In diesem Zusammenhang ergibt sich außerdem bereits die Optimierung des 175%-Slicings als mögliches Ziel. Bei diesem könnten etwa Feature-Modelle bei der Berechnung der Slice-Elemente miteinbezogen werden. Auf diese Weise würden die resultierenden Slices auch ohne Definition einer vollständigen Feature-Konfiguration im Slicing-Kriterium genauer und somit aussagekräftiger. Hierbei könnten etwa eine Menge von Feature-Modellen für die unterschiedlichen Versionen oder auch beispielsweise Temporale Feature-Modelle [31] in das Slicing eingebunden werden. Mit Hilfe dieser Feature-Modelle könnten Beziehungen zwischen den Features überprüft und korrekte Slices für partielle Feature-Konfigurationen als Kriterium erstellt werden.

Des Weiteren wäre es erstrebenswert das Zeitverhalten des Slicings zu verbessern, da vierzig Minuten Laufzeit für das Erstellen eines einfachen Slices nicht akzeptabel sind und eine adäquate Anwendung zur Analyse folglich annähernd ausgeschlossen ist. An dieser Stelle sollte bei der Erstellung des Abhängigkeitsgraphen angesetzt werden, da dieser mit steigender Komplexität der Modelle den wesentlichen Zeitanteil der Berechnung ausmacht.

Literatur

[1] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire und P. Merle. “Feature Model Differences”. In: *24th International Conference on Advanced Information Systems Engineering(CAiSE'12)*. LNCS. Springer, 06/2012, S. 629–645. URL: https://nyx.unice.fr/publis/acher-heymans-etal:2012.pdf.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba und C. Lucena. “Refactoring Product Lines”. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: ACM, 2006, S. 201–210. ISBN: 1-59593-237-2. DOI: 10.1145/1173706.1173737. URL: http://doi.acm.org/10.1145/1173706.1173737.

[3] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke und L. Tratt. “State-based Model Slicing: A Survey”. In: *ACM Comput. Surv.* 45.4 (08/2013), 53:1–53:36. ISSN: 0360-0300. DOI: 10.1145/2501654.2501667. URL: http://doi.acm.org/10.1145/2501654.2501667.

[4] D. Batory. “Feature Models, Grammars, and Propositional Formulas”. In: *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005. Proceedings*. Hrsg. von H. Obbink und K. Pohl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 7–20. ISBN: 978-3-540-32064-7. DOI: 10.1007/11554844_3. URL: http://dx.doi.org/10.1007/11554844_3.

[5] D. Benavides, S. Segura und A. Ruiz-Cortés. “Automated Analysis of Feature Models 20 Years Later: A Literature Review”. In: *Inf. Syst.* 35.6 (09/2010), S. 615–636. ISSN: 0306-4379. DOI: 10.1016/j.is.2010.01.001. URL: http://dx.doi.org/10.1016/j.is.2010.01.001.

[6] G. Botterweck und A. Pleuss. “Evolution of Software Product Lines”. In: *Evolving Software Systems*. Hrsg. von T. Mens, A. Serebrenik und A. Cleve. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 265–295. ISBN: 978-3-642-45398-4. DOI: 10.1007/978-3-642-45398-4_9. URL: http://dx.doi.org/10.1007/978-3-642-45398-4_9.

[7] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer und S. Kowalewski. “EvoFM: Feature-driven Planning of Product-line Evolution”. In: *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*. PLEASE '10. Cape Town, South Africa: ACM, 2010, S. 24–31. ISBN: 978-1-60558-968-8. DOI: 10.1145/1808937.1808941. URL: http://doi.acm.org/10.1145/1808937.1808941.

[8] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel und D. Beyer. “Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines”. In: *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Hrsg. von A. Egyed und I. Schaefer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, S. 84–99. ISBN: 978-3-662-46675-9. DOI: 10.1007/978-3-662-46675-9_6. URL: http://dx.doi.org/10.1007/978-3-662-46675-9_6.

[9] H. Cichos, S. Oster, M. Lochau und A. Schürr. "Model-Based Coverage-Driven Test Suite Generation for Software Product Lines". In: *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*. Hrsg. von J. Whittle, T. Clark und T. Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 425–439. ISBN: 978-3-642-24485-8. DOI: 10.1007/978-3-642-24485-8_31. URL: http://dx.doi.org/10.1007/978-3-642-24485-8_31.

[10] D. Clarke, M. Helvensteijn und I. Schaefer. "Abstract Delta Modeling". In: *SIGPLAN Not.* 46.2 (10/2010), S. 13–22. ISSN: 0362-1340. DOI: 10.1145/1942788.1868298. URL: http://doi.acm.org/10.1145/1942788.1868298.

[11] A. Classen. *Modelling with FTS: A Collection of Illustrative Examples*. Techn. Ber. P-CS-TR SPLMC-00000001. PReCISE Research Center, Univ. of Namur, 2010.

[12] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay und J.-F. Raskin. "Model checking lots of systems: efficient verification of temporal properties in software product lines". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM. 2010, S. 335–344.

[13] P. Clements und L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[14] K. Czarnecki und M. Antkiewicz. "Mapping features to models: A template approach based on superimposed variants". In: *International conference on generative programming and component engineering*. Springer. 2005, S. 422–437.

[15] D. Dhungana, P. Grünbacher, R. Rabiser und T. Neumayer. "Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering". In: *J. Syst. Softw.* 83.7 (07/2010), S. 1108–1122. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.02.018. URL: http://dx.doi.org/10.1016/j.jss.2010.02.018.

[16] K. Gallagher und D. Binkley. "Program slicing". In: *Frontiers of Software Maintenance, 2008. FoSM 2008*. IEEE. 2008, S. 58–67.

[17] A. Haber, H. Rendel, B. Rumpe und I. Schaefer. "Evolving Delta-Oriented Software Product Line Architectures". In: *Large-Scale Complex IT Systems. Development, Operation and Management: 17th Monterey Workshop 2012, Oxford, UK, March 19-21, 2012, Revised Selected Papers*. Hrsg. von R. Calinescu und D. Garlan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 183–208. ISBN: 978-3-642-34059-8. DOI: 10.1007/978-3-642-34059-8_10. URL: http://dx.doi.org/10.1007/978-3-642-34059-8_10.

[18] S. A. Hendrickson und A. van der Hoek. "Modeling Product Line Architectures Through Change Sets and Relationships". In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, S. 189–198. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.56. URL: http://dx.doi.org/10.1109/ICSE.2007.56.

[19] W. Ji, D. Wei und Q. Zhi-Chang. “Slicing Hierarchical Automata for Model Checking UML Statecharts”. In: *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21–25, 2002 Proceedings*. Hrsg. von C. George und H. Miao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 435–446. ISBN: 978-3-540-36103-9. DOI: 10.1007/3-540-36103-0_45. URL: http://dx.doi.org/10.1007/3-540-36103-0_45.

[20] J. Kamischke. “Entwurf und Implementierung eines Frameworks zum effizienten modellbasierten Testen von Software-Produktlinien”. Magisterarb. TU Braunschweig, 2012.

[21] J. Kamischke, M. Lochau und H. Baller. “Conditioned Model Slicing of Feature-annotated State Machines”. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. FOSD ’12. Dresden, Germany: ACM, 2012, S. 9–16. ISBN: 978-1-4503-1309-4. DOI: 10.1145/2377816.2377818. URL: http://doi.acm.org/10.1145/2377816.2377818.

[22] K. Kang, S. Cohen, J. Hess, W. Novak und A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Techn. Ber. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[23] G. Lima, J. Santos, U. Kulesza, D. A. da Costa und S. V. Fialho. “A Delta Oriented Approach to the Evolution and Reconciliation of Enterprise Software Products Lines”. In: *ICEIS 2013 - Proceedings of the 15th International Conference on Enterprise Information Systems, Volume 1, Angers, France, 4-7 July, 2013*. 2013, S. 255–263.

[24] S. Lity. “Konzeption und Evaluation eines delta-orientierten modellbasierten Testverfahrens für Softwareproduktlinien”. Magisterarb. TU Braunschweig, 2011.

[25] S. Lity, M. Kowal und I. Schaefer. “Higher-Order Delta Modeling for Software Product Line Evolution”. In: *Proceedings of the International Workshop on Feature-Oriented Software Development*. FOSD ’16. Eindhoven, The Netherlands: ACM, 2016. ISBN: 978-1-4503-0208-1. DOI: 10.1145/1868688.1868696. URL: http://doi.acm.org/10.1145/1868688.1868696.

[26] S. Lity, R. Lachmann, M. Lochau und I. Schaefer. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. Techn. Ber. 2012-07. Technische Universität Braunschweig, 2012.

[27] J. McGregor. *The Evolution of Product Line Assets*. Techn. Ber. CMU/SEI-2003-TR-005. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6601.

[28] T. Mens, A. Serebrenik und A. Cleve, Hrsg. *Evolving Software Systems*. Springer, 2014. ISBN: 978-3-642-45397-7. DOI: 10.1007/978-3-642-45398-4. URL: http://dx.doi.org/10.1007/978-3-642-45398-4.

[29] S. Nahrendorf. *Entwicklung und Modellierung von Evolutionsszenarien für Delta-orientierte Softwareproduktlinien: Projektarbeit*. Braunschweig, 2017. DOI: 10.24355/dbbs.084-201704071225. URL: https://dx.doi.org/10.24355/dbbs.084-201704071225.

[30] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena und U. Kulesza. "Safe Evolution Templates for Software Product Lines". In: *J. Syst. Softw.* 106.C (08/2015), S. 42–58. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.04.024. URL: http://dx.doi.org/10.1016/j.jss.2015.04.024.

[31] M. Nieke, C. Seidl und S. Schuster. "Guaranteeing Configuration Validity in Evolving Software Product Lines". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, S. 73–80. ISBN: 978-1-4503-4019-9. DOI: 10.1145/2866614.2866625. URL: http://doi.acm.org/10.1145/2866614.2866625.

[32] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer und S. Kowalewski. "Model-driven Support for Product Line Evolution on Feature Level". In: *J. Syst. Softw.* 85.10 (10/2012), S. 2261–2274. ISSN: 0164-1212. DOI: 10.1016/j.jss.2011.08.008. URL: http://dx.doi.org/10.1016/j.jss.2011.08.008.

[33] K. Pohl, G. Böckle und F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN: 3540243720.

[34] I. Schaefer. "Variability Modelling for Model-Driven Development of Software Product Lines". In: *VAMOS*. 2010.

[35] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo und K. Villela. "Software Diversity: State of the Art and Perspectives". In: *STTT* 14.5 (2012), S. 477–495. ISSN: 1433-2779.

[36] C. Seidl, I. Schaefer und U. Aßmann. "Capturing Variability in Space and Time with Hyper Feature Models". In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 6:1–6:8. ISBN: 978-1-4503-2556-1. DOI: 10.1145/2556624.2556625. URL: http://doi.acm.org/10.1145/2556624.2556625.

[37] C. Seidl, I. Schaefer und U. Aßmann. "Integrated Management of Variability in Space and Time in Software Families". In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. SPLC '14. Florence, Italy: ACM, 2014, S. 22–31. ISBN: 978-1-4503-2740-4. DOI: 10.1145/2648511.2648514. URL: http://doi.acm.org/10.1145/2648511.2648514.

[38] F. Tip. *A Survey of Program Slicing Techniques*. Techn. Ber. Amsterdam, The Netherlands, The Netherlands, 1994.

[39] M. Weiser. "Program Slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, S. 439–449. ISBN: 0-89791-146-6. URL: http://dl.acm.org/citation.cfm?id=800078.802557.

A Verwendung der Eclipse Plug-ins

In diesem Anhang soll erklärt werden, wie die in dieser Arbeit erstellten Eclipse Plug-ins verwendet werden können.

Es wird empfohlen das Eclipse-Package *Eclipse Modeling Tools* zu verwenden, damit alle notwendigen Plug-ins des Eclipse Modeling Framework bereits vorhanden sind. Zur Erprobung der neu implementierten Plug-ins wurde beispielsweise die aktuelle Eclipse-Version *Oxygen*[1] benutzt.

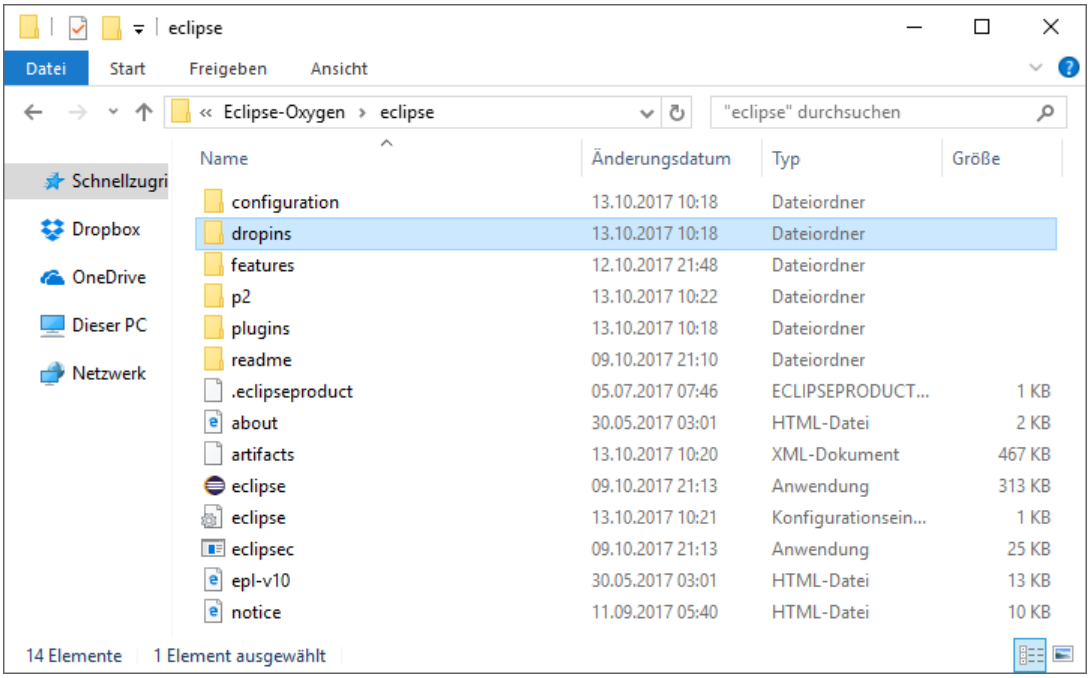


Abbildung A.1.: Ordnerstruktur Eclipse

Nachdem das Archiv, welches Eclipse enthält, entpackt wurde, findet sich in den Unterordnern des Eclipse-Ordners der Ordner *dropins* (s. Abbildung A.1). In diesen Ordner werden die Plug-in-Dateien für die Projekte, welche auf dem Datenträger im Ordner *Implementierung* gespeichert sind, eingefügt. Hierbei handelt es sich um mehr als die in Kapitel 5 beschriebenen Plug-ins, da einige der beschriebenen Plug-ins wiederum Abhängigkeiten zu den hier gelisteten Plug-ins aufweisen.

Anschließend kann Eclipse ganz normal gestartet und die in dieser Arbeit vorgestellten Funktionalitäten, wie beispielsweise der Editor für 175%-Modelle oder die Versionsableitung, können

[1] http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/oxygen1a, Stand 15.10.2017

über *New* > *Other...* verwendet werden. Sollen bestimmte Funktionalitäten der Plug-ins, wie etwa der Transformationsalgorithmus oder die Versionsableitung, in anderen Projekten genutzt werden, müssen die entsprechenden Plug-ins per Rechtsklick auf den Projektordner unter *Build Path* > *Configure Build Path...* in das gewünschte Projekt eingebunden werden. Dazu werden die Plugins unter *Libraries* > *Add External JARs...* aus dem Ordner *Plugin* herausgesucht und zum Pfad hinzugefügt. So können dann beispielsweise sämtliche, in Kapitel 5.3 beschriebene Methoden aus der Klasse `RunDerivation` zur Versionsableitung eingesetzt werden.

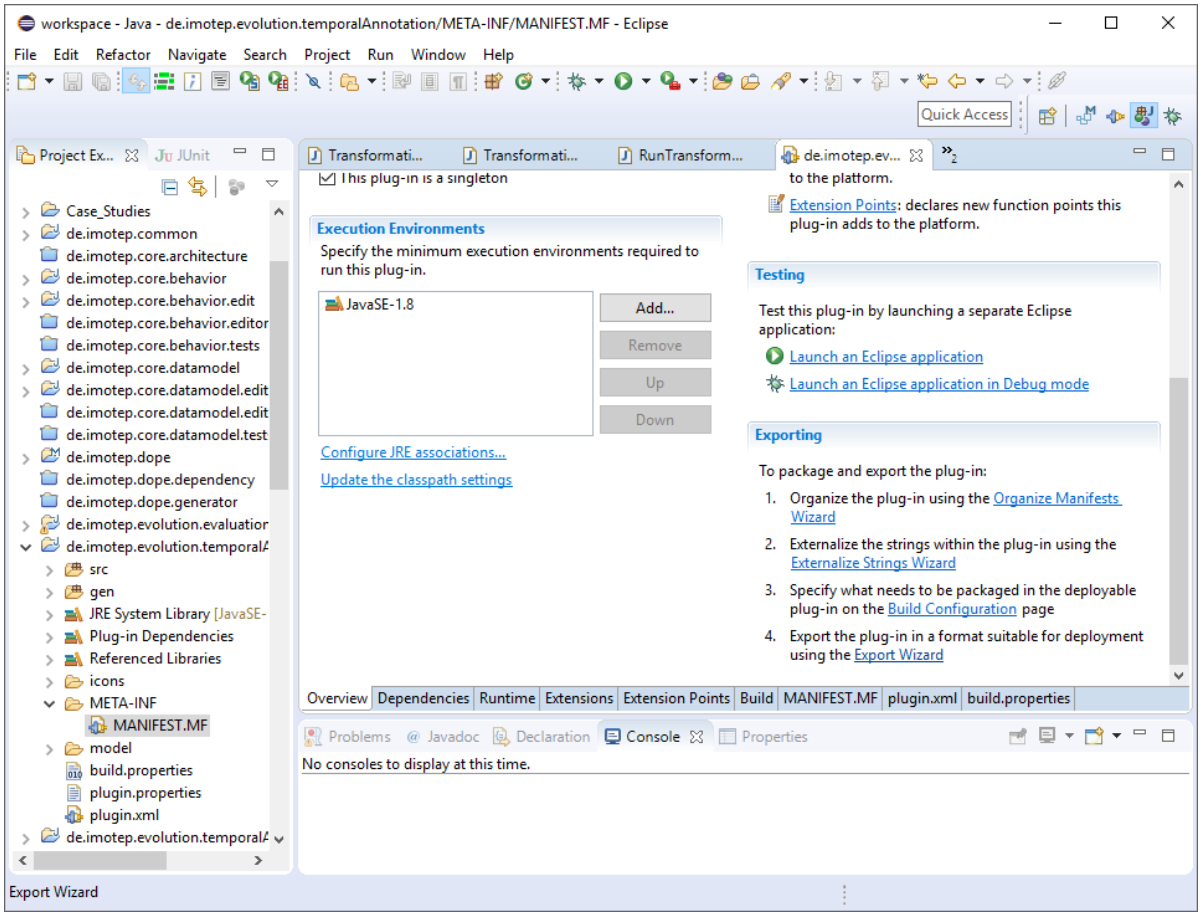


Abbildung A.2.: Export eines Plug-ins

Die Plug-ins können aus den auf dem Datenträger befindlichen Projekten erzeugt werden, indem die `MANIFEST.MF`-Datei des jeweiligen Projekts aufgerufen wird. Wie in Abbildung A.2 abgebildet, kann dort unter *Overview* > *Exporting* > *Export Wizard* ein Plug-in für das gewählte Projekt erzeugt werden.

B Inhalt des Datenträgers

Die beigefügte CD enthält die folgenden Inhalte:

- `/Dokument/`: LaTeX-Dateien und Abbildungen dieser Masterarbeit
- `/Fallstudien/`: Modelle der Fallstudien (Higher-Order Delta-Modelle, 175%-Modelle, 150%-Modell-Versionen)
- `/Implementierung/`: Quellcode der realisierten Eclipse Plug-ins
  - `Abhängige Projekte/`: Projekte, zu denen der Quellcode der Eclipse Plug-ins Abhängigkeiten besitzt
  - `Evaluation/`: Für die Evaluation verwendete, zusätzliche Implementierung
- `/Masterarbeit.pdf`: PDF-Dokument der Ausarbeitung

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende Projektarbeit ohne Hilfe Dritter und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Technische Universität Carolo-Wilhelmina in Braunschweig
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23
D-38106 Braunschweig